# ECE 471 Lab 3

MD5 Collision Attack Lab

Mitchell Dzurick

3/9/2020

**Github with all documentation -** `https://www.github.com/mitchdz/ECE471`

## Contents

Secret Key Encryption Lab

## 1 Overview

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 hash(M ) = h. The collision-resistance property ensures that it is compu- tationally infeasible to find two different inputs M 1 and M 2 , such that hash(M 1 ) = hash(M 2 ). Several widely-used one-way hash functions have trouble maintaining the collision-resistance prop- erty. At the rump session of CRYPTO 2004, Xiaoyun Wang and co-authors demonstrated a collision attack against MD5 [1]. In February 2017, CWI Amsterdam and Google Research announced the SHAttered attack, which breaks the collision-resistance property of SHA-1 [3]. While many students do not have trouble understanding the importance of the one-way property, they cannot 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 first hand 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
- The collision-resistance property
- Collision attacks
- MD5

**Lab Environment**. This lab has been tested on our pre-built Ubuntu 12.04 VM and Ubuntu 16.04 VM, both of which can be downloaded from the SEED website.

## 2 Lab Tasks

## 3 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
```

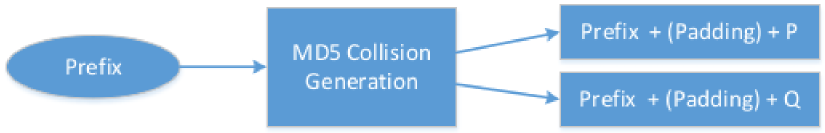


Figure 1: Running the key generation program without srand multiple times

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.

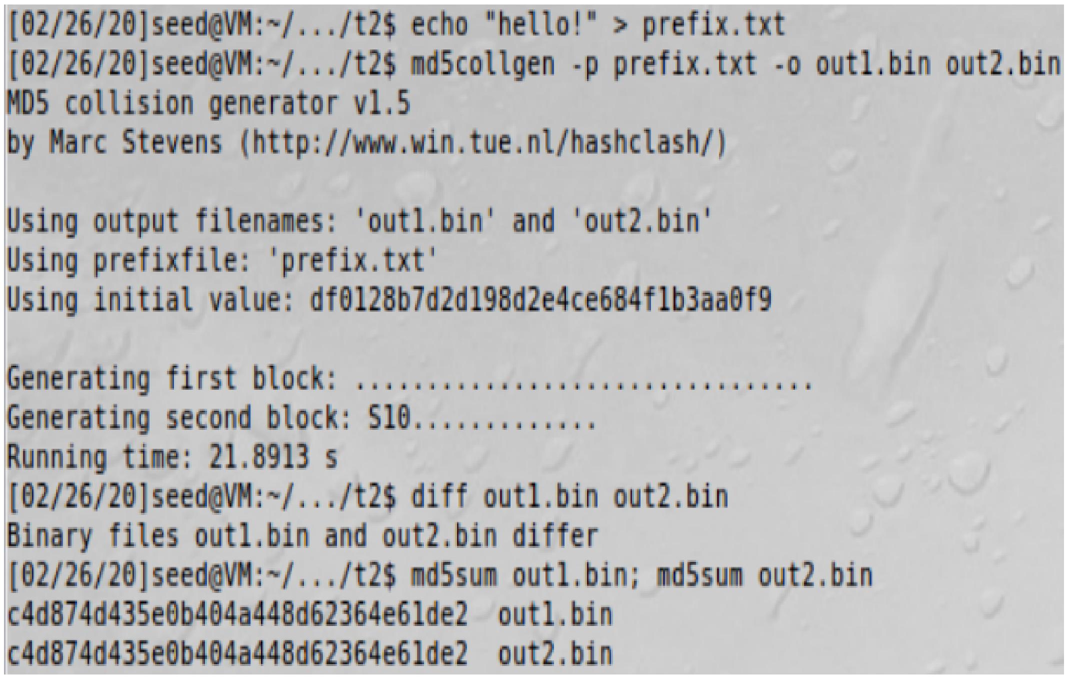
```
$ diff out1.bin out2.bin
$ md5sum out1.bin
$ md5sum out2.bin
```

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 bless 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.

## 3.1 Task 1 Solution

Initial observations are required in the md5collgen program. To do this, let's execute the program.



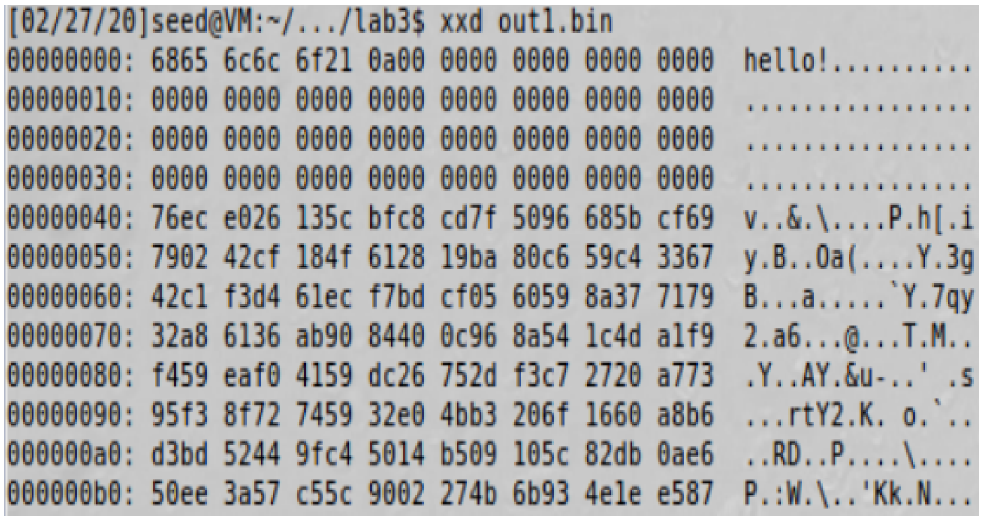
```
[02/26/20]seed@VM:~/.../t2$ echo "hello!" > prefix.txt
[02/26/20]seed@VM:~/.../t2$ md5collgen -p prefix.txt -o out1.bin out2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1.bin' and 'out2.bin'
Using prefixfile: 'prefix.txt'
Using initial value: df0128b7d2d198d2e4ce684f1b3aa0f9

Generating first block: ..............................
Generating second block: S10............
Running time: 21.8913 s
[02/26/20]seed@VM:~/.../t2$ diff out1.bin out2.bin
Binary files out1.bin and out2.bin differ
[02/26/20]seed@VM:~/.../t2$ md5sum out1.bin; md5sum out2.bin
c4d874d435e0b404a448d62364e61de2  out1.bin
c4d874d435e0b404a448d62364e61de2  out2.bin
```

Figure 2: investigating md5collgen program

As Figure 2 shows, a file with the contents ”hello!” being thrown into md5collgen. The output is two files, named ”out1.bin” and ”out2.bin”. These hash of these files are then produced using the commands md5sum. It can be seen that the files are different through the ‘diff‘ command, but their md5sum hash is the same.

### 3.1.1 Task 1 Question 1 Solution

the prefix was padded with zero bytes until the size is a multiple of 64.



```
[02/27/20]seed@VM:~/.../lab3$ xxd out1.bin
00000000: 6865 6c6c 6f21 0a00 0000 0000 0000 0000  hello!..........
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 76ec e026 135c bfc8 cd7f 5096 685b cf69  v..&.\....P.h[.i
00000050: 7902 42cf 184f 6128 19ba 80c6 59c4 3367  y.B..Oa(....Y.3g
00000060: 42c1 f3d4 61ec f7bd cf05 6059 8a37 7179  B...a.....`Y.7qy
00000070: 32a8 6136 ab90 8440 0c96 8a54 1c4d a1f9  2.a6...@...T.M..
00000080: f459 eaf0 4159 dc26 752d f3c7 2720 a773  .Y..AY.&u-..' .s
00000090: 95f3 8f72 7459 32e0 4bb3 206f 1660 a8b6  ...rtY2.K. o.`..
000000a0: d3bd 5244 9fc4 5014 b509 105c 82db 0ae6  ..RD..P....\....
000000b0: 50ee 3a57 c55c 9002 274b 6b93 4e1e e587  P.:W.\..'Kk.N...
```

Figure 3: file not of 64 bytes being ran through md5collgen

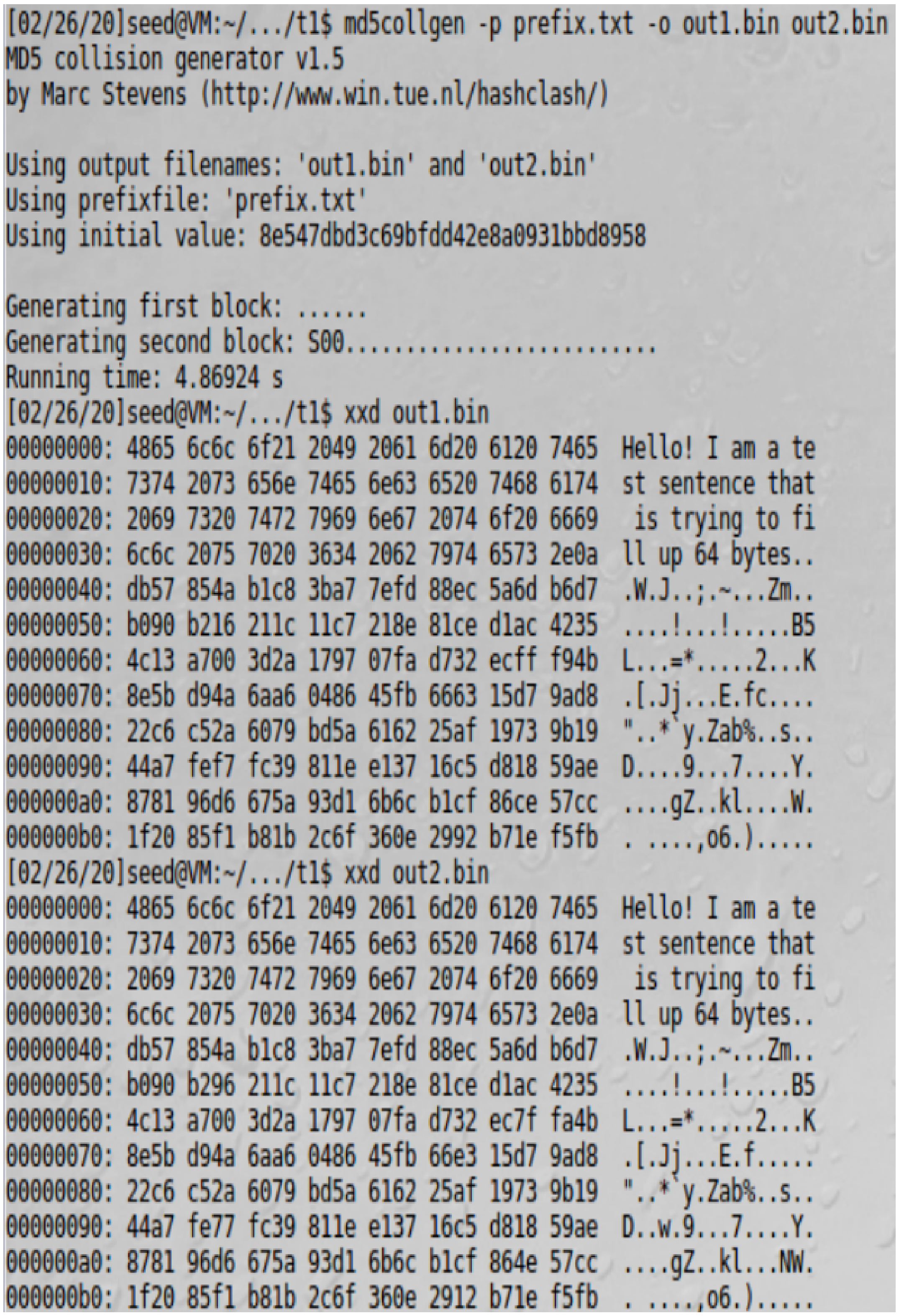
Figure 3 shows the results from Figure 2. out1.bin is shown to be padded with 0's until 64 bytes, then seemingly random data is appended.

### 3.1.2 Task 1 Question 2 Solution

With 64-byte prefix, no bytes are needed for padding. The prefix then has extra data added on for the collision. Figure 4 shows these results as there is no extra 0 bytes added, but rather just extra data to aid in the collision.

### 3.1.3 Task 1 Question 3 Solution

A file named prefix.txt is created with contents "Hello! I am a test sentence that is trying to fill up 64 bytes." this file is then ran through the md5collgen.



```
[02/26/20]seed@VM:~/.../t1$ 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: 8e547dbd3c69bfdd42e8a0931bbd8958

Generating first block: ......
Generating second block: S00.........................
Running time: 4.86924 s
[02/26/20]seed@VM:~/.../t1$ xxd out1.bin
00000000: 4865 6c6c 6f21 2049 2061 6d20 6120 7465  Hello! I am a te
00000010: 7374 2073 656e 7465 6e63 6520 7468 6174  st sentence that
00000020: 2069 7320 7472 7969 6e67 2074 6f20 6669   is trying to fi
00000030: 6c6c 2075 7020 3634 2062 7974 6573 2e0a  ll up 64 bytes..
00000040: db57 854a b1c8 3ba7 7efd 88ec 5a6d b6d7  .W.J..;.~...Zm..
00000050: b090 b216 211c 11c7 218e 81ce d1ac 4235  ....!...!.....B5
00000060: 4c13 a700 3d2a 1797 07fa d732 ecff f94b  L...=*.....2...K
00000070: 8e5b d94a 6aa6 0486 45fb 6663 15d7 9ad8  .[.Jj...E.fc....
00000080: 22c6 c52a 6079 bd5a 6162 25af 1973 9b19  "..*`y.Zab%..s..
00000090: 44a7 fef7 fc39 811e e137 16c5 d818 59ae  D....9...7....Y.
000000a0: 8781 96d6 675a 93d1 6b6c b1cf 86ce 57cc  ....gZ..kl....W.
000000b0: 1f20 85f1 b81b 2c6f 360e 2992 b71e f5fb  . ....,o6.).....
[02/26/20]seed@VM:~/.../t1$ xxd out2.bin
00000000: 4865 6c6c 6f21 2049 2061 6d20 6120 7465  Hello! I am a te
00000010: 7374 2073 656e 7465 6e63 6520 7468 6174  st sentence that
00000020: 2069 7320 7472 7969 6e67 2074 6f20 6669   is trying to fi
00000030: 6c6c 2075 7020 3634 2062 7974 6573 2e0a  ll up 64 bytes..
00000040: db57 854a b1c8 3ba7 7efd 88ec 5a6d b6d7  .W.J..;.~...Zm..
00000050: b090 b296 211c 11c7 218e 81ce d1ac 4235  ....!...!.....B5
00000060: 4c13 a700 3d2a 1797 07fa d732 ec7f fa4b  L...=*.....2...K
00000070: 8e5b d94a 6aa6 0486 45fb 66e3 15d7 9ad8  .[.Jj...E.f.....
00000080: 22c6 c52a 6079 bd5a 6162 25af 1973 9b19  "..*`y.Zab%..s..
00000090: 44a7 fe77 fc39 811e e137 16c5 d818 59ae  D..w.9...7....Y.
000000a0: 8781 96d6 675a 93d1 6b6c b1cf 864e 57cc  ....gZ..kl...NW.
000000b0: 1f20 85f1 b81b 2c6f 360e 2912 b71e f5fb  . ....,o6.).....
```

Figure 4: 64 bytes of prefix being put into the md5collgen

Figure 4 shows that the files don't completely differ, but they do differ in certain areas. It's actually apparent that only 8 bytes are different out of the 192 bytes that are output. The bytes that changed only changed a little bit as well. They only differ in one bit.

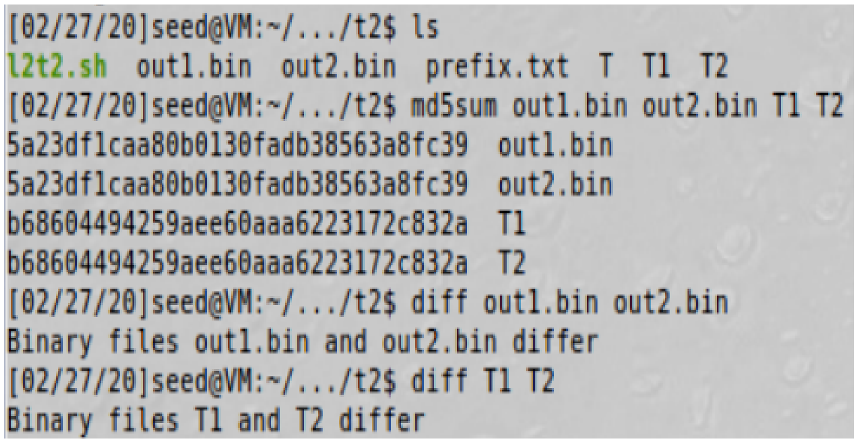
## 3.2 Task 2: Understanding MD5's Property

### 3.2.1 Task 2: Solution

In order to finish this task, some setup is required. To generate all of the files, the commands below are executed.

```
echo "hello!" > prefix.txt
md5collgen -p prefix.txt -o out1.bin out2.bin
echo "T" > T
cat out1.bin T > T1
cat out2.bin T > T2
```

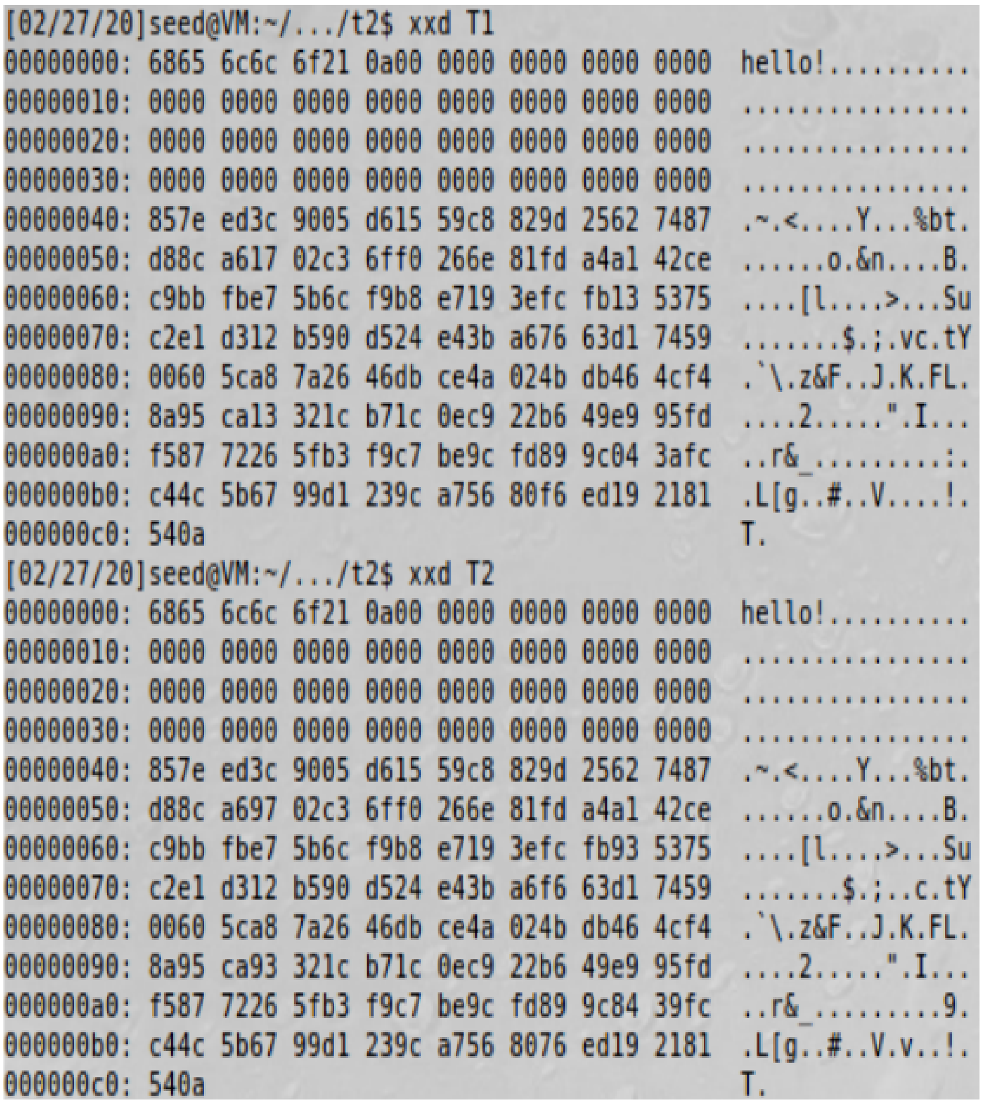
These commands will generate the following files: T, out1.bin, out2.bin, T1, T2. T is a file that contains the hex value ”0x540a” because of the *echo ”T” ¿ T* command. out1.bin and out2.bin are the files generated by md5collgen which their hash values match as shown in Figure 5.



```
[02/27/20]seed@VM:~/.../t2$ ls
l2t2.sh  out1.bin  out2.bin  prefix.txt  T  T1  T2
[02/27/20]seed@VM:~/.../t2$ md5sum out1.bin out2.bin T1 T2
5a23df1caa80b0130fadb38563a8fc39  out1.bin
5a23df1caa80b0130fadb38563a8fc39  out2.bin
b68604494259aee60aaa6223172c832a  T1
b68604494259aee60aaa6223172c832a  T2
[02/27/20]seed@VM:~/.../t2$ diff out1.bin out2.bin
Binary files out1.bin and out2.bin differ
[02/27/20]seed@VM:~/.../t2$ diff T1 T2
Binary files T1 and T2 differ
```

Figure 5: matching md5sum values

Figure 5 shows the results of running all of the aforementioned commands. It can be seen that the md5 value of out1.bin and out2.bin match, and the md5sum value of T1 and T2 also match. Below that, the command ‘diff‘ is used to show that the binary value of every file is different, yet they produce the same hash.



```
[02/27/20]seed@VM:~/.../t2$ xxd T1
00000000: 6865 6c6c 6f21 0a00 0000 0000 0000 0000  hello!..........
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 857e ed3c 9005 d615 59c8 829d 2562 7487  .~.<....Y...%bt.
00000050: d88c a617 02c3 6ff0 266e 81fd a4a1 42ce  ......o.&n....B.
00000060: c9bb fbe7 5b6c f9b8 e719 3efc fb13 5375  ....[l....>...Su
00000070: c2e1 d312 b590 d524 e43b a676 63d1 7459  .......$.;.vc.tY
00000080: 0060 5ca8 7a26 46db ce4a 024b db46 4cf4  .`\.z&F..J.K.FL.
00000090: 8a95 ca13 321c b71c 0ec9 22b6 49e9 95fd  ....2.....".I...
000000a0: f587 7226 5fb3 f9c7 be9c fd89 9c04 3afc  ..r&_.........:.
000000b0: c44c 5b67 99d1 239c a756 80f6 ed19 2181  .L[g..#..V....!.
000000c0: 540a                                     T.
[02/27/20]seed@VM:~/.../t2$ xxd T2
00000000: 6865 6c6c 6f21 0a00 0000 0000 0000 0000  hello!..........
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 857e ed3c 9005 d615 59c8 829d 2562 7487  .~.<....Y...%bt.
00000050: d88c a697 02c3 6ff0 266e 81fd a4a1 42ce  ......o.&n....B.
00000060: c9bb fbe7 5b6c f9b8 e719 3efc fb93 5375  ....[l....>...Su
00000070: c2e1 d312 b590 d524 e43b a6f6 63d1 7459  .......$.;..c.tY
00000080: 0060 5ca8 7a26 46db ce4a 024b db46 4cf4  .`\.z&F..J.K.FL.
00000090: 8a95 ca93 321c b71c 0ec9 22b6 49e9 95fd  ....2.....".I...
000000a0: f587 7226 5fb3 f9c7 be9c fd89 9c84 39fc  ..r&_.........9.
000000b0: c44c 5b67 99d1 239c a756 8076 ed19 2181  .L[g..#..V.v..!.
000000c0: 540a                                     T.
```

Figure 6: hex values of output

Figure 6 also shows the hex values of T1 and T2 which shows that 0x540a is appended to each file.

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

### 3.3.1 Task 3: Solution

This section explores how to create two different executable files with the same has. The idea behind this task is that you have a binary executable with a section you want to change. assuming the prefix and the suffix are the same, we know that MD5(prefix || P || suffix) = MD5(prefix || Q || suffix) where P and Q is data generated from md5collgen.

For this task, a program needs to be compiled such that an array has changed values, but the hash of the executables are the same. The program is shown below.

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

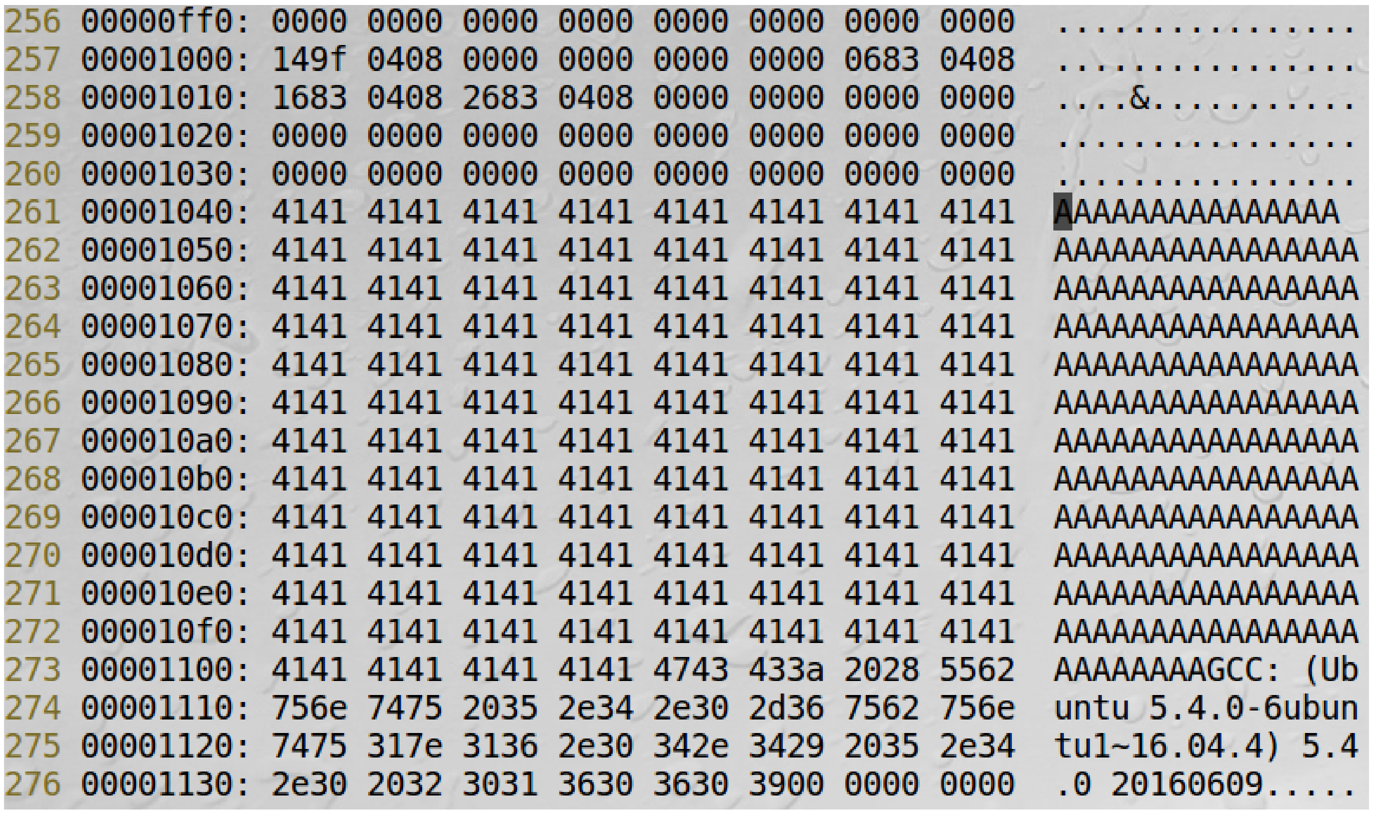
The executable is generated with with gcc.

```
$ gcc prog.c
```

using vim, we can view the raw hex of the outputted file, a.out

```
vim a.out
```

Then the command ':%!xxd' is ran inside vim to launch xxd within vim. Using vim, we search for the string of A's with the command '/AAAA' and find the following:



```
256 00000ff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
257 00001000: 149f 0408 0000 0000 0000 0000 0683 0408  ................
258 00001010: 1683 0408 2683 0408 0000 0000 0000 0000  ....&...........
259 00001020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
260 00001030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
261 00001040: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
262 00001050: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
263 00001060: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
264 00001070: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
265 00001080: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
266 00001090: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
267 000010a0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
268 000010b0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
269 000010c0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
270 000010d0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
271 000010e0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
272 000010f0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
273 00001100: 4141 4141 4141 4141 4743 433a 2028 5562  AAAAAAAAGCC: (Ub
274 00001110: 756e 7475 2035 2e34 2e30 2d36 7562 756e  untu 5.4.0-6ubun
275 00001120: 7475 317e 3136 2e30 342e 3429 2035 2e34  tu1~16.04.4) 5.4
276 00001130: 2e30 2032 3031 3630 3630 3900 0000 0000  .0 20160609.....
```

Figure 7: raw hex of a.out

The first line of A' begin on line 261 (byte 4176 because 261 * 16). We want a multiple of 16 so we choose byte 4224 which is line 264. Let's retrieve all of the binary data before byte 4224 with the following command

```
$ head -c 4224 a.out > prefix
```

we then have the prefix that we can use the md5collgen program on.

```
$ md5collgen -p prefix -o P Q
```

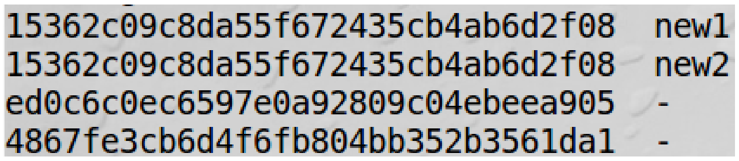
Finally, we need to extract the rest of the data in the program. Let's grab 128 bytes after the start of the line we chose, 4224 + 128 = 4352.

```
$ tail -c +4352 a.out > suffix
```

Now we have all of the components to build the different executables but with the same hash!

```
$ cat P suffix > new1
$ cat Q suffix > new2
$ md5sum new1 new2
$ chmod +x new1
$ chmod +x new2
$ echo $(./new1) | md5sum
$ echo $(./new2) | md5sum
```

The output of the previous command is shown below.



```
15362c09c8da55f672435cb4ab6d2f08  new1
15362c09c8da55f672435cb4ab6d2f08  new2
ed0c6c0ec6597e0a92809c04ebeea905  -
4867fe3cb6d4f6fb804bb352b3561da1  -
```

Figure 8: md5hash of new files

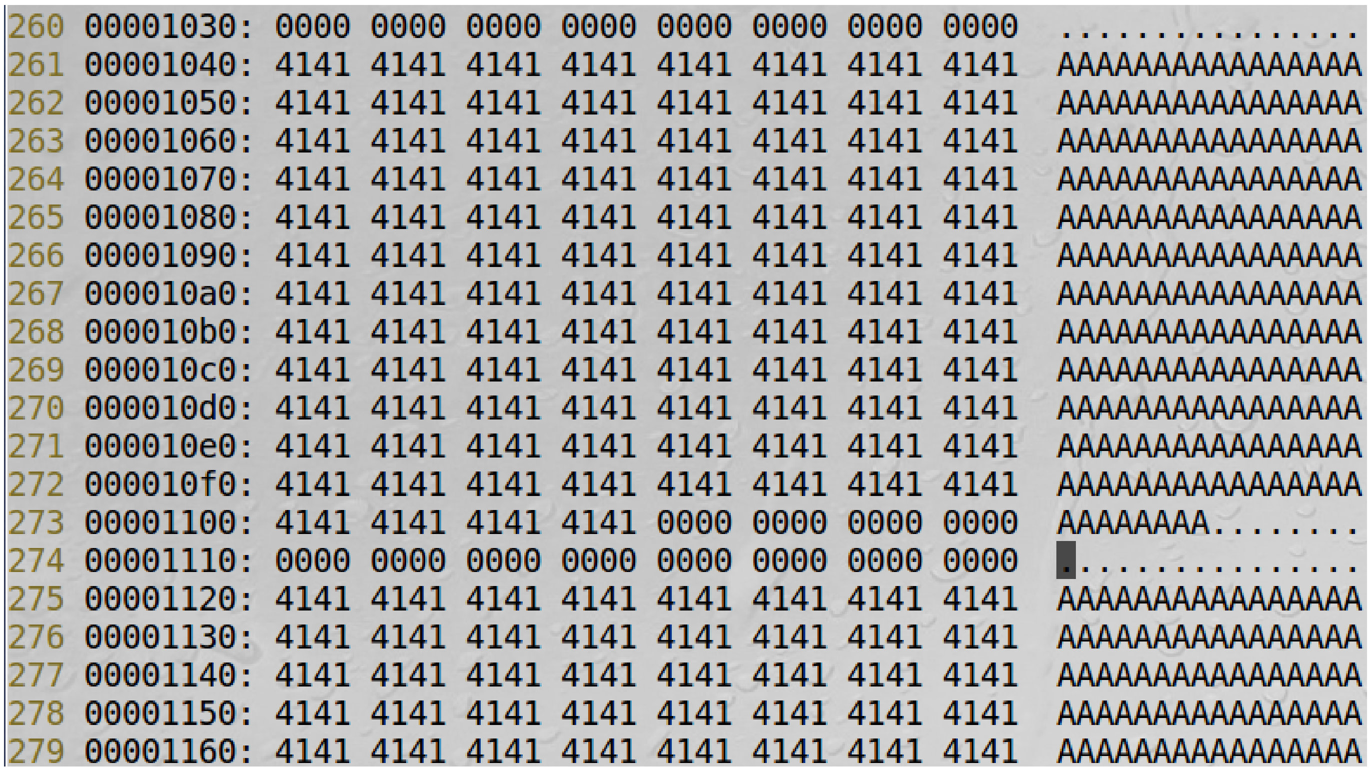
Figure 9 shows the results of running md5sum on the executable files new1 and new2. These files are then executed and their output is hashes as well for simplicity in comparison. It is very easy to determine that the output of the files are different from their hash.

## 3.4 Task 4: Making the Two Programs Behave Differently

### 3.4.1 Task 4: Solution

In order to complete this task, some modification to the code in Task 3 is needed. The following resultant code is shown below:

```
#include <stdio.h>

unsigned char b[200] = {<200 A's>};
unsigned char b[200] = {<200 A's>};
int main()
{
int flag = 1;

    for (int i = 0;i<200;i++) {
        if (a[i] != b[1]){flag = 0; break;}
    }
    if(flag) printf("Regular execution!\n");
    else printf("Malicious code!\n");
    return 0;
}
```



```
260 00001030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
261 00001040: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
262 00001050: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
263 00001060: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
264 00001070: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
265 00001080: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
266 00001090: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
267 000010a0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
268 000010b0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
269 000010c0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
270 000010d0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
271 000010e0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
272 000010f0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
273 00001100: 4141 4141 4141 4141 0000 0000 0000 0000  AAAAAAAA........
274 00001110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
275 00001120: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
276 00001130: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
277 00001140: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
278 00001150: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
279 00001160: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
```

Figure 9: md5hash of new files

running gcc on the program as follows:

```
$ gcc prog.c
```

produces the executable a.out which we can view the hex in vim by using the keystroke ':%!xxd' once in vim. Figure **??** shows the results of viewing the file's hex with vim. It can be seen that the first batch of 'AAAAAAAAAAAAAAAAA's begins on line 261 which is byte 4176. We want a multiple of 64 so let's use line 262 which is byte 4192.

The following code is used to create the two executables:

```
START="4192"
END=$(($START + 129))

#extract the prefix and temporary suffix
head -c ${START} a.out > prefix
md5collgen -p prefix -o out1 out2
tail -c +${END} a.out > suffixtemp

# take the first 8 bytes of the suffix and add them to the end of out1 and out2
head -c 8 suffixtemp > arrtemp
cat out1 arrtemp > out1arr
cat out2 arrtemp > out2arr
```

```
tail -c +9 suffixtemp > suffix

# now we need to fix the buffer between the two arrays
tail -c +25 suffix > suffixtemp
head -c 24 suffix > buffer
cat out1arr buffer > file1buffer
cat out2arr buffer > file2buffer

# now the malicious stuff happens
tail -c +201 suffixtemp > suffix
tail -c +4161 out1arr > comparray
cat file1buffer comparray suffix > goodCode
cat file2buffer comparray suffix > maliciousCode

md5sum goodCode maliciousCode
chmod +x goodCode
chmod +x maliciousCode

./goodCode
./maliciousCode
```

Upon executing the above code, the following is output:

```
Regular execution!
Malicious code!
```

Thus showcasing how to exploit this particular vulnerability