CRACKING MD5

4-20-23
CIS 4362 Applied Cryptography

Justine Saylor, Veronica Spitnale,
Georgette Amancha
Professor Mukhopadhyay

Problem Definition

MD5, which stands for Message Digest 5, is a hash algorithm that produces a 128-bit hash. The MD5 algorithm first starts by padding bits to the original input string in order for the size of the string to be 64 bits short of a multiple of 512. Next, the goal is to get the message length to be an exact multiple of 512 bits. In order for this to happen the original message length has to be in a 64-bit representation which is then appended to the result after the padding bits were added. Next, the final resultant string of 128-bits is divided into four 32-bit variables that are initialized to the fixed constants A, B, C, and D. Then, each 512-bit block is broken down and goes through four rounds of operations. The first part of our problem was to use existing MD5 hash lookup tables to crack hashes. Hash lookup tables store the key-to-value paths and allow users to lookup a given hash and recover the password. The second part of our problem was to launch a collision attack on MD5. One of the main weaknesses of MD5 is it is prone to hash collisions which means it is possible to create the same hash output for two different inputs.

Brief Literature About MD5

MD5 was created as an improved version of MD4. Although MD5 is similar to MD4 in that they both produce 128-bit hashes and have a similar design, MD5 is more complex in the execution of its algorithm. One of the main improvements in MD5 is it has a fourth round of operations in its main loop.

For example, for the fixed constant A, the variables B, C, and D are passed into a non-linear function. The result of this function is added to the value of A, a unique constant, and a sub-block value. The result is then rotated to the right a variable number of bits, added to the value of B, and replaces the previous value of A. This process described for A, is run for B, C, and D and for every sub-block. The main loop for MD5 is pictured on the next page to better visual the process.

The four nonlinear functions:

$$F(X,Y,Z) = (X \wedge Y) \vee ((\sim X) \wedge Z)$$
$$G(X,T,Z) = (X \wedge Z) \vee (Y \wedge (\sim Z))$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee (\sim Z))$$

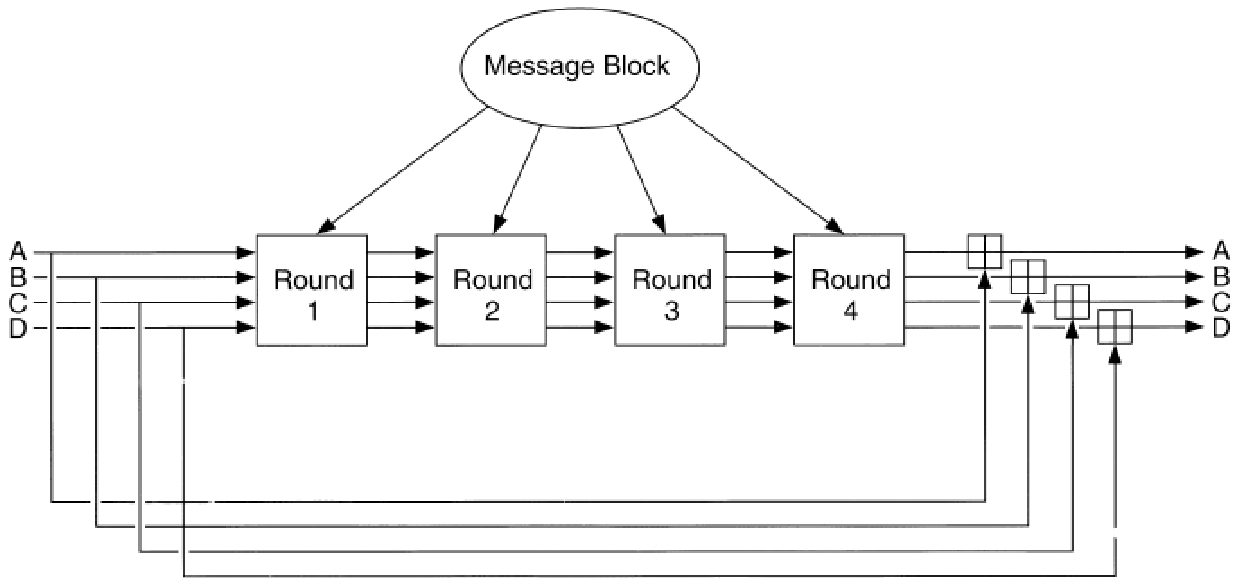
($\oplus$ is XOR, $\wedge$ is AND, $\vee$ is OR, and ~ is NOT).

While MD5 has some weaknesses such as being prone to hash collisions, slow, and still being one-way it is still widely used today due to its advantages. Some of these advantages include being easier to compare since it is a smaller hash, uses low resource consumption, and can be used for both storing passwords and checking integrity. Hash functions are mainly used for password and integrity verification since the same output is always provided for the same input.

Our Approach

Our approach for cracking hashes using lookup tables started with downloading a password cracking dictionary. This dictionary contains 64 million lines of passwords obtained from database leaks. The next step was we created a function to hash the passwords from the text file containing all the passwords using an open source (cited in works cited) MD5 algorithm code. Next, we output the hashed password and plaintext into an output file to read from. The code then prompts the user to input a MD5 hash, which then prompts the code to search through the output file for the matching hatch. If the matching hash is found, the password in plaintext is output to the console. Otherwise, a message stating the hash was not found is displayed on the console.

Our approach for launching a collision attack on MD5 started with downloading and setting up the Oracle VM Virtual Box and Ubuntu. We used these software systems to run a MD5 collision attack lab. Our first step was to use the md5collegen program to create two files with the same prefix and MD5 hash values. Next, we used these two files to demonstrate how if two inputs have the same hash and suffix then the two outputs will result in the same hash value. Next, given a C program we created two different versions. These versions had different array contents but the MD5 hash values were the same. Finally, we created two programs where if the contents of two arrays were the same then benign code was executed, otherwise malicious code was executed. This generated two different versions with different contents that still had the same MD5 hash values.

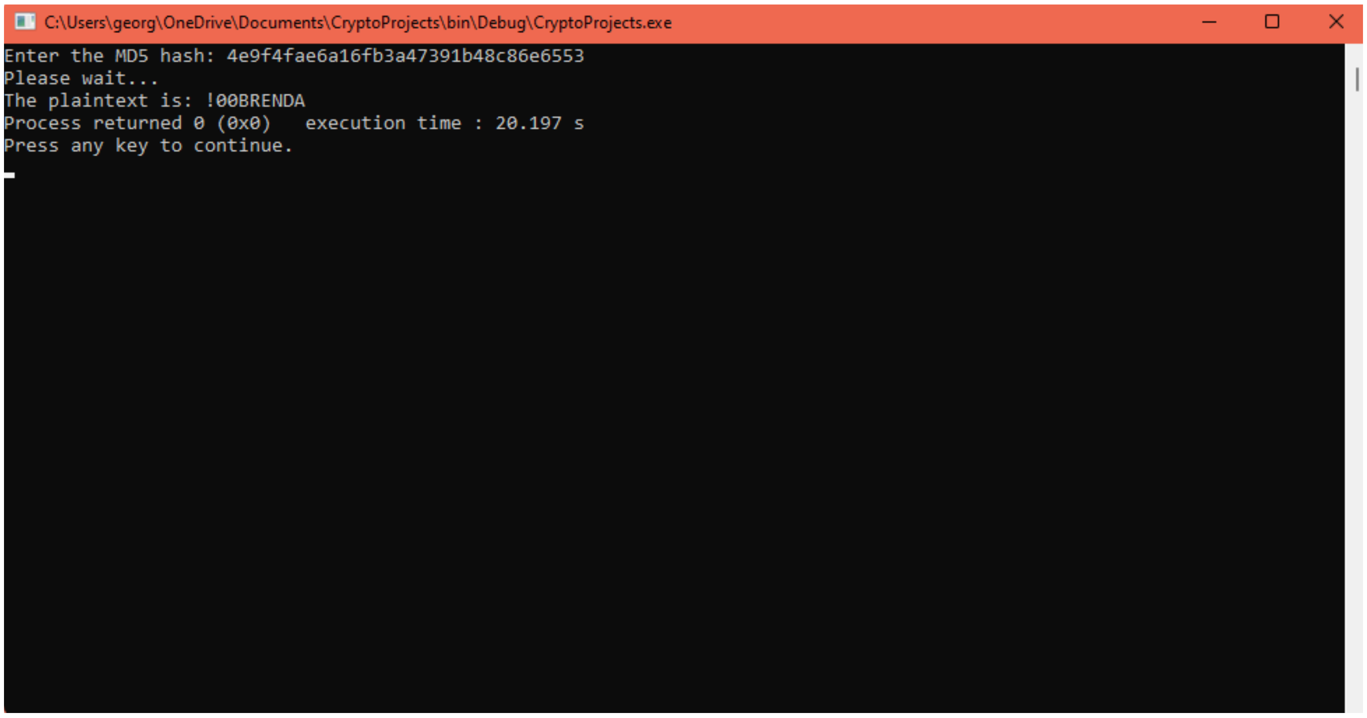


Results Analysis

The first part of this project involved using existing MD5 hash lookup tables to crack hashes. By completing this part of the project, we were able to understand what hashing is and specifically how the MD5 algorithm works. By inputting a string, the MD5 function can then be used to create a 128-bit digest. From this knowledge we were able to create a code that demonstrates how the original password or string can be computed based on the hash value. This result occurred because of the property of MD5 in which each input always maps to the same output.

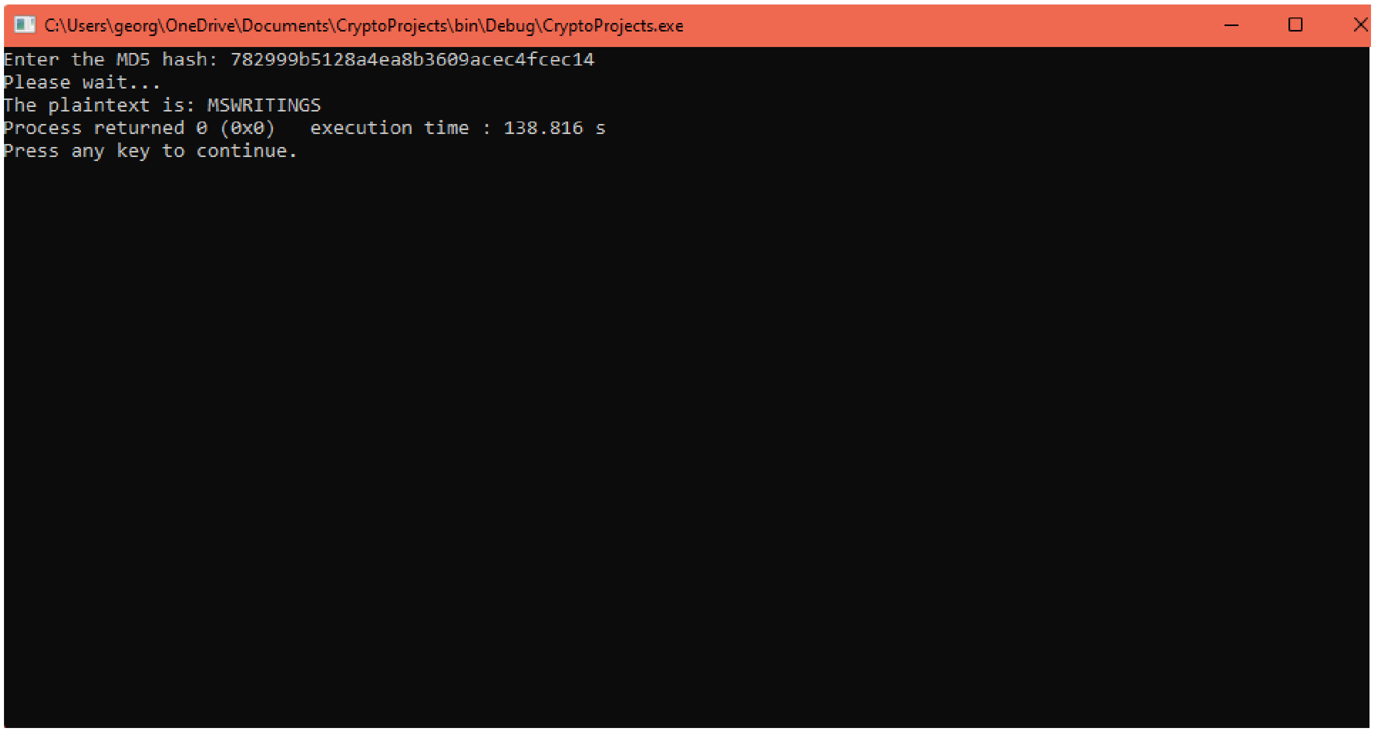
The second part of this project involved launching a collision attack on MD5. By completing this part of the project, we were able to understand the major weakness of the MD5 algorithm where hash collisions can occur. A hash collision occurs when two different inputs map to the same output. This was demonstrated in our project when two files with different contents both had the same hash value. This part of the project showed us how problematic hash collisions can be when it comes to using MD5 for integrity and password verification. From the first task in the collision lab, we also saw how the MD5 algorithm will pad bits to the file prefix if it is not a multiple of 64 by comparing how the program ran with a file prefix that was exactly 64 bytes to a file prefix that was not a multiple of 64.

Results for Part A (Cracking Hashes Using Lookup Tables):



C:\Users\georg\OneDrive\Documents\CryptoProjects\bin\Debug\CryptoProjects.exe

```
Enter the MD5 hash: 4e9f4fae6a16fb3a47391b48c86e6553
Please wait...
The plaintext is: !00BRENDA
Process returned 0 (0x0)   execution time : 20.197 s
Press any key to continue.
```



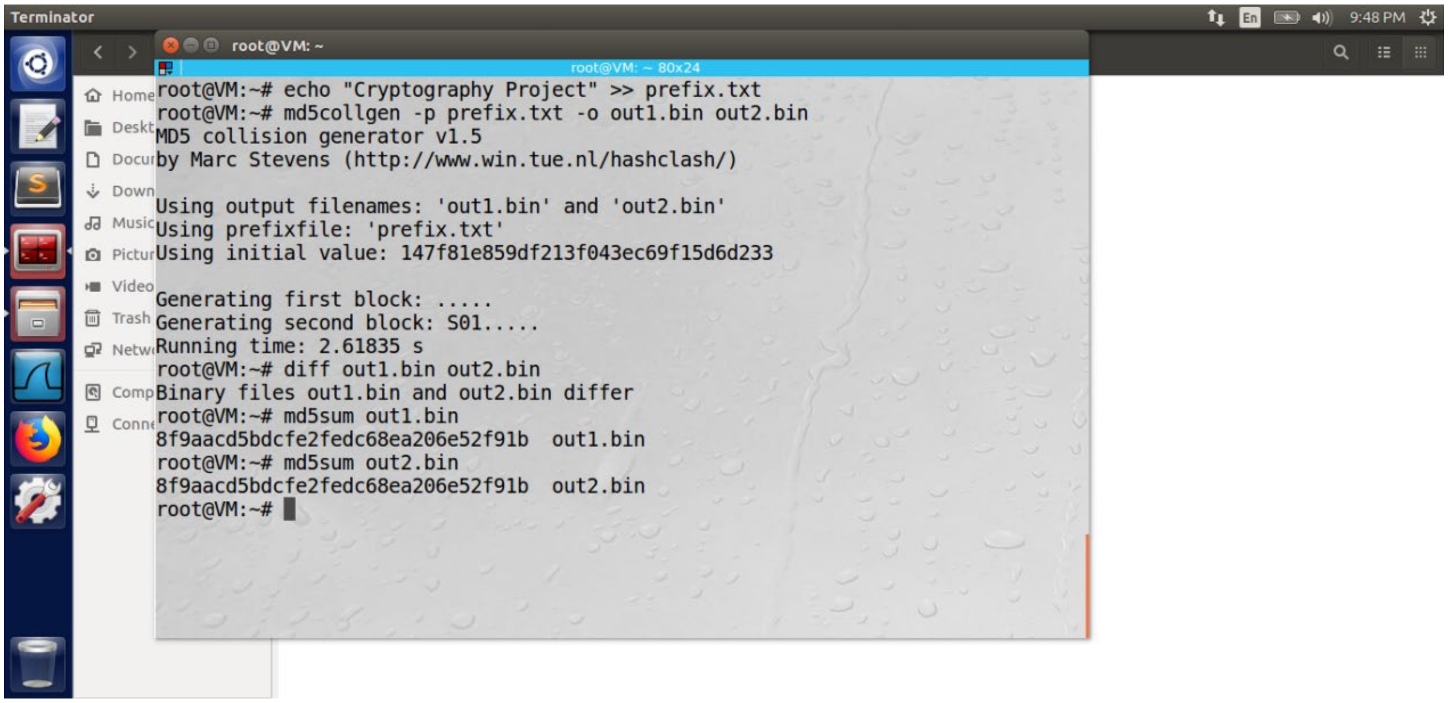
C:\Users\georg\OneDrive\Documents\CryptoProjects\bin\Debug\CryptoProjects.exe

```
Enter the MD5 hash: 782999b5128a4ea8b3609acec4fcec14
Please wait...
The plaintext is: MSWRITINGS
Process returned 0 (0x0)   execution time : 138.816 s
Press any key to continue.
```

These images show how after inputting two different MD5 hashes, they each have their own output plaintext and the execution time it took to find the password.

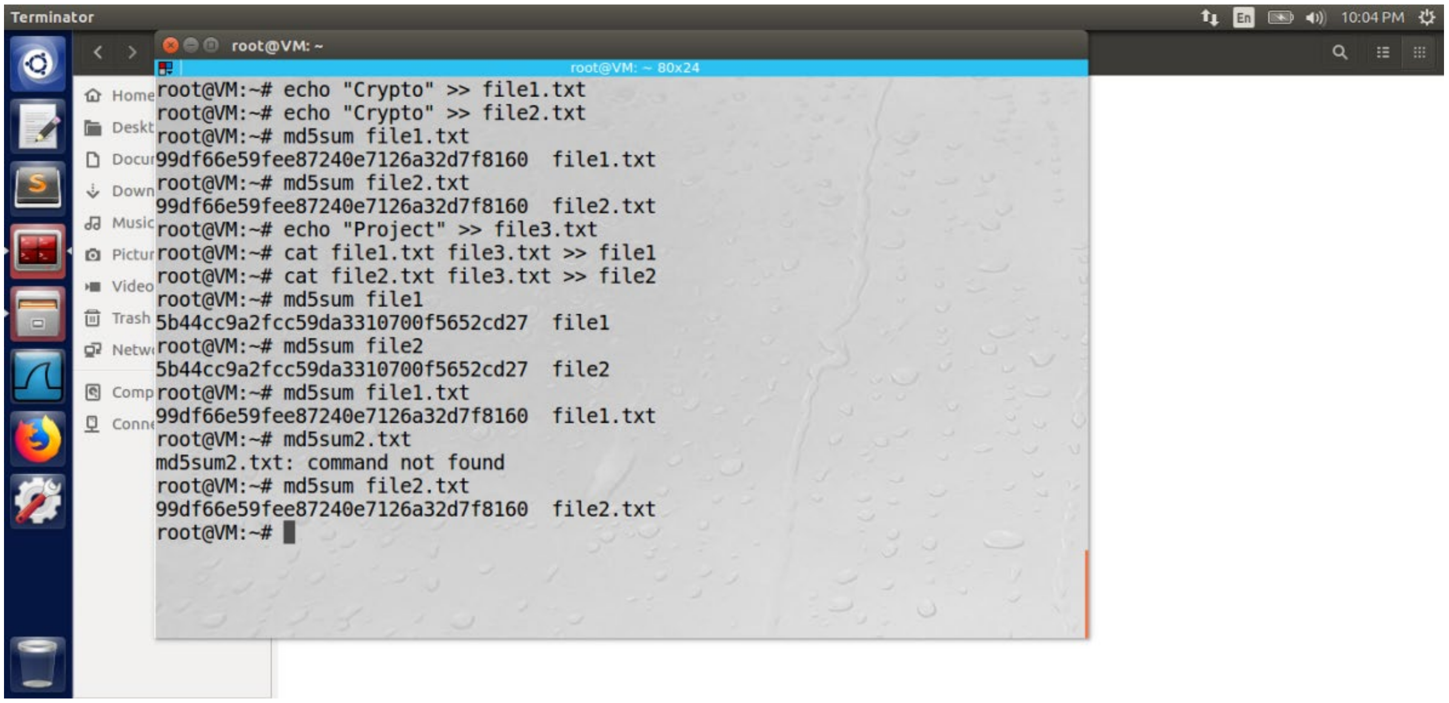
* code for part a is in separate zip file*

Results and Code for Part B (Collision Attack):



```
root@VM:~# echo "Cryptography Project" >> prefix.txt
root@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: 147f81e859df213f043ec69f15d6d233

Generating first block: .....
Generating second block: S01.....
Running time: 2.61835 s
root@VM:~# diff out1.bin out2.bin
Binary files out1.bin and out2.bin differ
root@VM:~# md5sum out1.bin
8f9aacd5bdcfe2fedc68ea206e52f91b  out1.bin
root@VM:~# md5sum out2.bin
8f9aacd5bdcfe2fedc68ea206e52f91b  out2.bin
root@VM:~# 
```

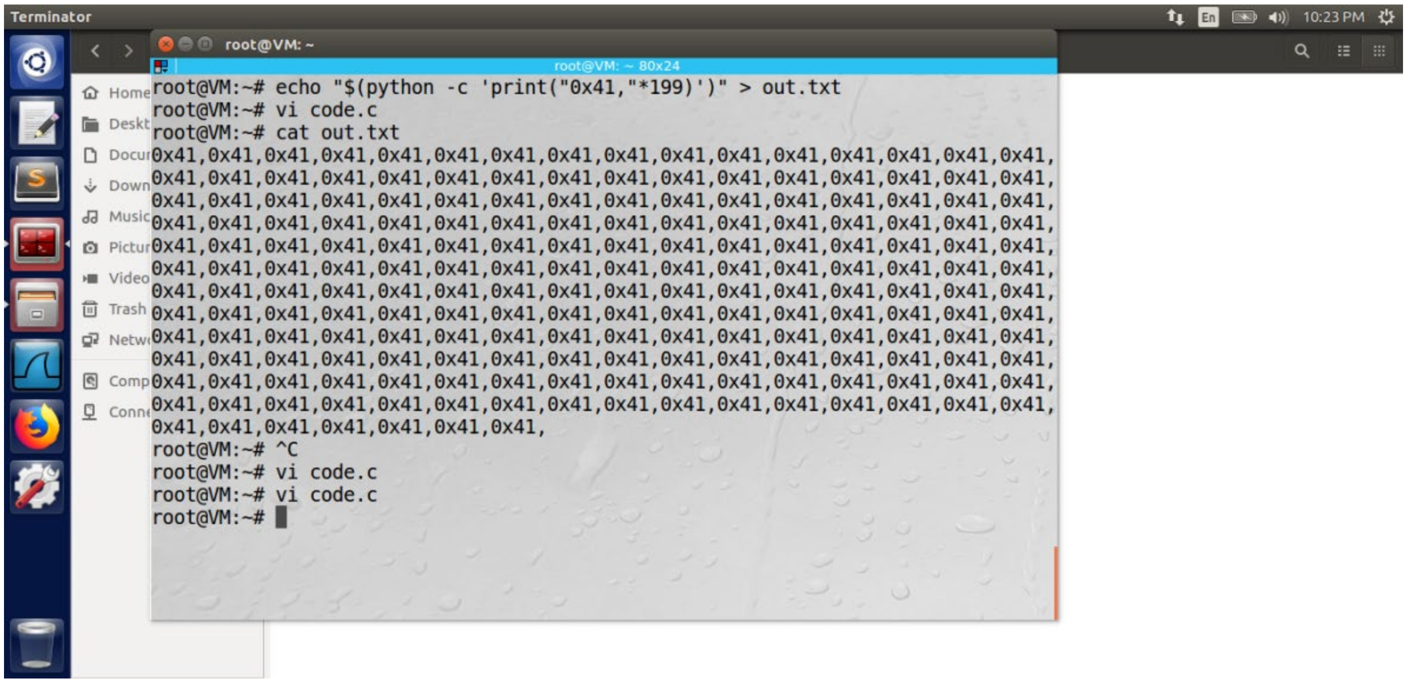
Task 1 – This image shows two output files, out1.bin and out2.bin, being created using the given prefix file prefix.txt. It also shows how even though both of these files were created with the same prefix file, they differ which was shown by using the diff command.



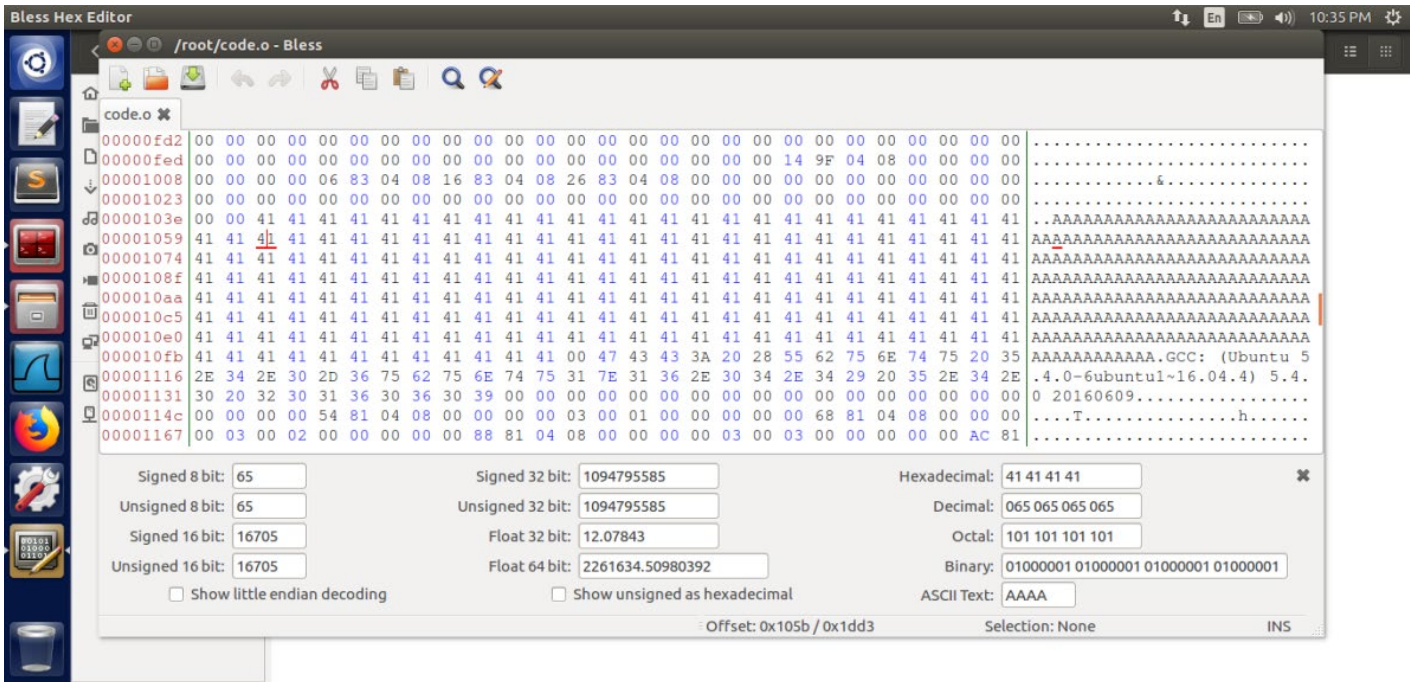
```
root@VM:~# echo "Crypto" >> file1.txt
root@VM:~# echo "Crypto" >> file2.txt
root@VM:~# md5sum file1.txt
99df66e59fee87240e7126a32d7f8160  file1.txt
root@VM:~# md5sum file2.txt
99df66e59fee87240e7126a32d7f8160  file2.txt
root@VM:~# echo "Project" >> file3.txt
root@VM:~# cat file1.txt file3.txt >> file1
root@VM:~# cat file2.txt file3.txt >> file2
root@VM:~# md5sum file1
5b44cc9a2fcc59da3310700f5652cd27  file1
root@VM:~# md5sum file2
5b44cc9a2fcc59da3310700f5652cd27  file2
root@VM:~# md5sum file1.txt
99df66e59fee87240e7126a32d7f8160  file1.txt
root@VM:~# md5sum2.txt
md5sum2.txt: command not found
root@VM:~# md5sum file2.txt
99df66e59fee87240e7126a32d7f8160  file2.txt
root@VM:~# 
```

Task 2 – This image shows how if two inputs have the same hash and you add the same suffix, then both outputs will have the same hash value. This was done by using the cat command to concatenate the contents of file 3 to file 1 and file 2.

Results and Code for Part B (Collision Attack):



```
root@VM:~# echo "$(python -c 'print("0x41,"*199)')" > out.txt
root@VM:~# vi code.c
root@VM:~# cat out.txt
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
0x41,0x41,0x41,0x41,0x41,0x41,0x41,
root@VM:~# ^C
root@VM:~# vi code.c
root@VM:~# vi code.c
root@VM:~#
```



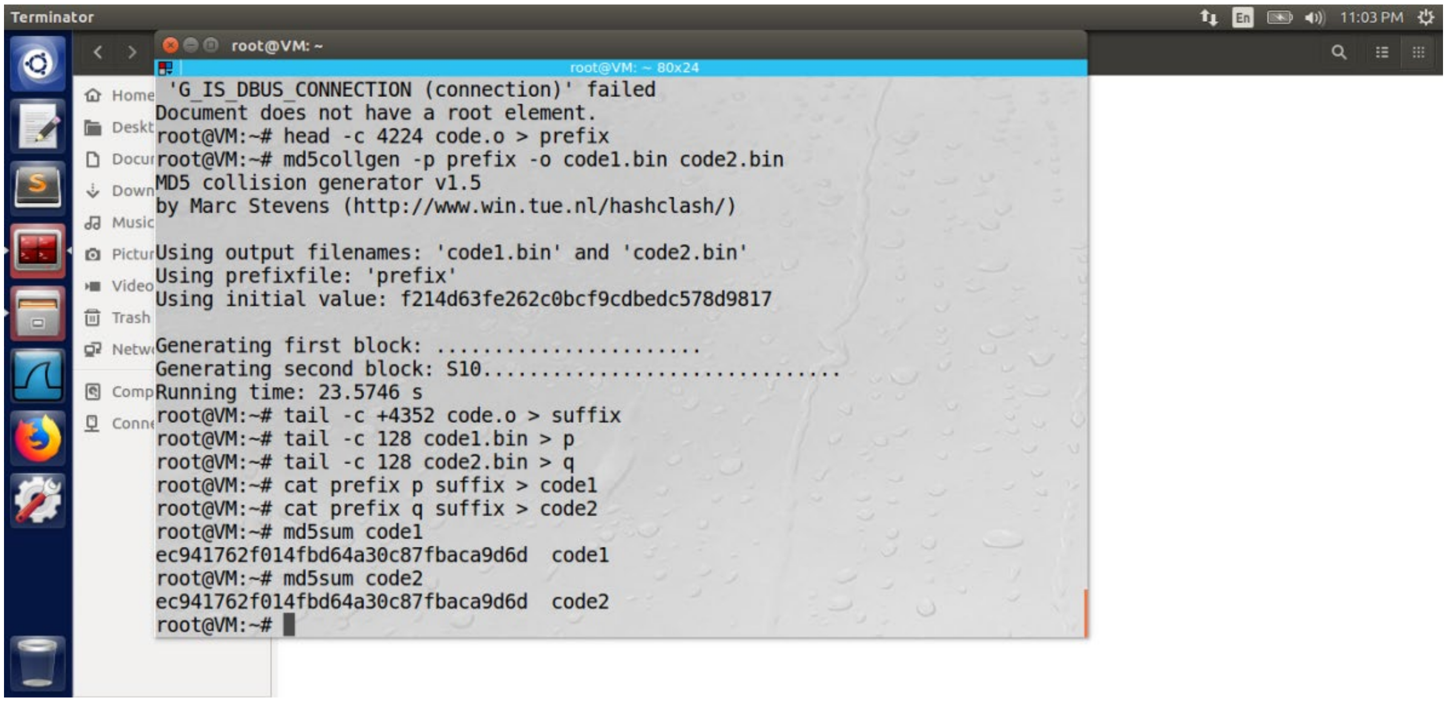
/root/code.o - Bless

```
00000fd2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...........................
00000fed 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 14 9F 04 08 00 00 00 00 ...........................
00001008 00 00 00 00 06 83 04 08 16 83 04 08 26 83 04 08 00 00 00 00 00 00 00 00 00 00 00 ............&..............
00001023 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...........................
0000103e 00 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 ..AAAAAAAAAAAAAAAAAAAAAAAAA
00001059 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAA
00001074 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAA
0000108f 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAA
000010aa 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAA
000010c5 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAA
000010e0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAAAAAA
000010fb 41 41 41 41 41 41 41 41 41 41 41 41 00 47 43 43 3A 20 28 55 62 75 6E 74 75 20 35 AAAAAAAAAAAA.GCC: (Ubuntu 5
00001116 2E 34 2E 30 2D 36 75 62 75 6E 74 75 31 7E 31 36 2E 30 34 2E 34 29 20 35 2E 34 2E .4.0-6ubuntu1~16.04.4) 5.4.
00001131 30 20 32 30 31 36 30 36 30 39 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0 20160609.................
0000114c 00 00 00 00 54 81 04 08 00 00 00 00 03 00 01 00 00 00 00 00 68 81 04 08 00 00 00 ....T...............h......
00001167 00 03 00 02 00 00 00 00 00 88 81 04 08 00 00 00 00 03 00 03 00 00 00 00 00 AC 81 ...........................
```

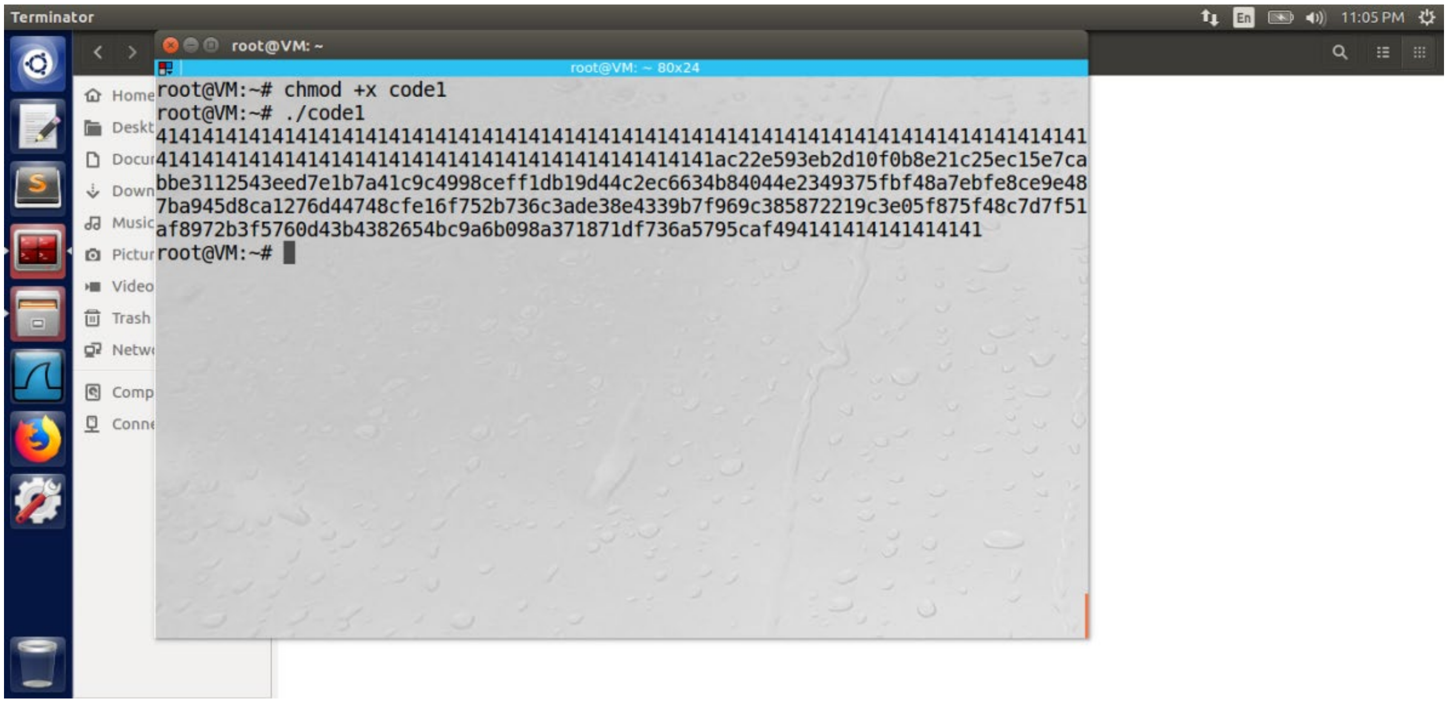
```
   Signed 8 bit: 65            Signed 32 bit: 1094795585         Hexadecimal: 41 41 41 41
 Unsigned 8 bit: 65          Unsigned 32 bit: 1094795585             Decimal: 065 065 065 065
  Signed 16 bit: 16705          Float 32 bit: 12.07843                 Octal: 101 101 101 101
Unsigned 16 bit: 16705          Float 64 bit: 2261634.50980392        Binary: 01000001 01000001 01000001 01000001
[ ] Show little endian decoding   [ ] Show unsigned as hexadecimal    ASCII Text: AAAA
                                  Offset: 0x105b / 0x1dd3   Selection: None   INS
```

Task 3 – The first images shows an array that was filled with 0x41 to make it easier to find the contents of the array in image 2. In image 2 on the right hand side, you can see a bunch of As since 0x41 is the ASCII value for the letter A making it fast to find where to modify the contents of the array.

Results and Code for Part B (Collision Attack):



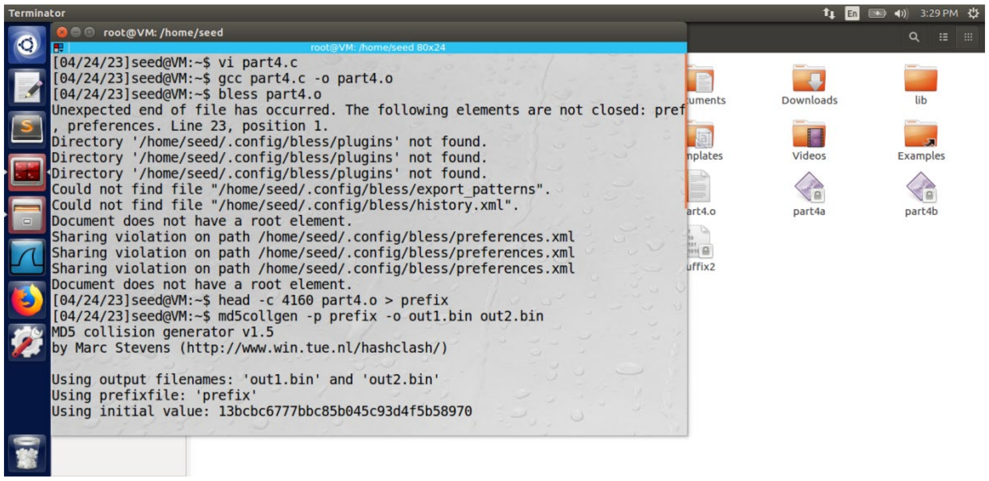
```
'G_IS_DBUS_CONNECTION (connection)' failed
Document does not have a root element.
root@VM:~# head -c 4224 code.o > prefix
root@VM:~# md5collgen -p prefix -o code1.bin code2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'code1.bin' and 'code2.bin'
Using prefixfile: 'prefix'
Using initial value: f214d63fe262c0bcf9cdbedc578d9817

Generating first block: ......................
Generating second block: S10..............................
Running time: 23.5746 s
root@VM:~# tail -c +4352 code.o > suffix
root@VM:~# tail -c 128 code1.bin > p
root@VM:~# tail -c 128 code2.bin > q
root@VM:~# cat prefix p suffix > code1
root@VM:~# cat prefix q suffix > code2
root@VM:~# md5sum code1
ec941762f014fbd64a30c87fbaca9d6d  code1
root@VM:~# md5sum code2
ec941762f014fbd64a30c87fbaca9d6d  code2
root@VM:~# 
```



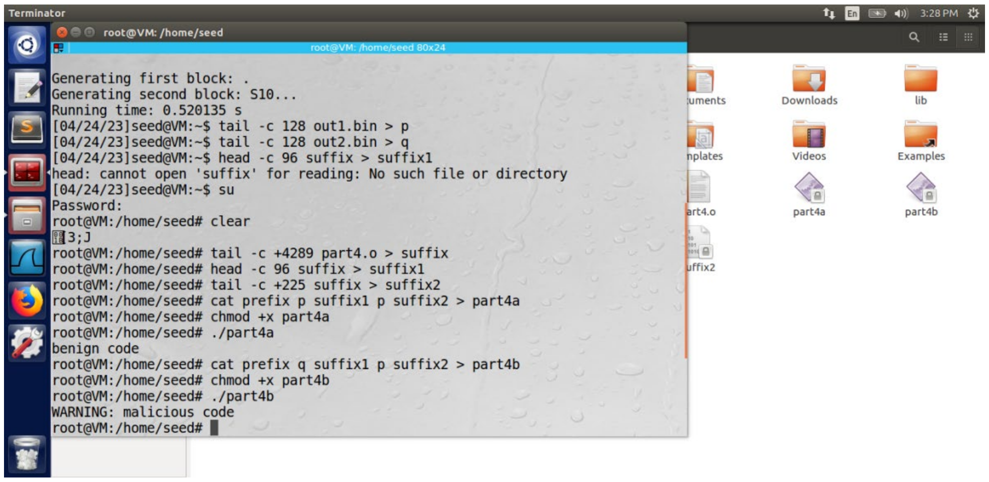
```
root@VM:~# chmod +x code1
root@VM:~# ./code1
41414141414141414141414141414141414141414141414141414141414141414141414141414141
414141414141414141414141414141414141414141414141ac22e593eb2d10f0b8e21c25ec15e7ca
bbe3112543eed7e1b7a41c9c4998ceff1db19d44c2ec6634b84044e2349375fbf48a7ebfe8ce9e48
7ba945d8ca1276d44748cfe16f752b736c3ade38e4339b7f969c385872219c3e05f875f48c7d7f51
af8972b3f5760d43b4382654bc9a6b098a371871df736a5795caf494141414141414141
root@VM:~# 
```

Task 3 Continued – The first image shows the prefix values P and Q being used to replace the 128 bytes of the array, therefore creating two binary programs that have the same hash value but different contents in their own arrays. The second image displays the hash value for code 1 which is the same as code 2.

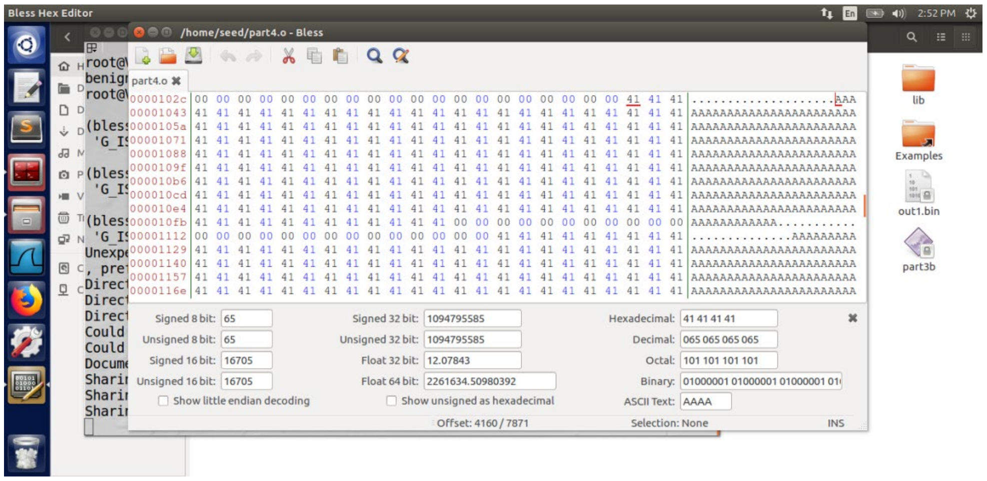
Results and Code for Part B (Collision Attack):



```
[04/24/23]seed@VM:~$ vi part4.c
[04/24/23]seed@VM:~$ gcc part4.c -o part4.o
[04/24/23]seed@VM:~$ bless part4.o
Unexpected end of file has occurred. The following elements are not closed: pref
, preferences. Line 23, position 1.
Directory '/home/seed/.config/bless/plugins' not found.
Directory '/home/seed/.config/bless/plugins' not found.
Directory '/home/seed/.config/bless/plugins' not found.
Could not find file "/home/seed/.config/bless/export_patterns".
Could not find file "/home/seed/.config/bless/history.xml".
Document does not have a root element.
Sharing violation on path /home/seed/.config/bless/preferences.xml
Sharing violation on path /home/seed/.config/bless/preferences.xml
Sharing violation on path /home/seed/.config/bless/preferences.xml
Document does not have a root element.
[04/24/23]seed@VM:~$ head -c 4160 part4.o > prefix
[04/24/23]seed@VM:~$ md5collgen -p prefix -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'
Using initial value: 13bcbc6777bbc85b045c93d4f5b58970
```



```
Generating first block: .
Generating second block: S10...
Running time: 0.520135 s
[04/24/23]seed@VM:~$ tail -c 128 out1.bin > p
[04/24/23]seed@VM:~$ tail -c 128 out2.bin > q
[04/24/23]seed@VM:~$ head -c 96 suffix > suffix1
head: cannot open 'suffix' for reading: No such file or directory
[04/24/23]seed@VM:~$ su
Password:
root@VM:/home/seed# clear
3;J
root@VM:/home/seed# tail -c +4289 part4.o > suffix
root@VM:/home/seed# head -c 96 suffix > suffix1
root@VM:/home/seed# tail -c +225 suffix > suffix2
root@VM:/home/seed# cat prefix p suffix1 p suffix2 > part4a
root@VM:/home/seed# chmod +x part4a
root@VM:/home/seed# ./part4a
benign code
root@VM:/home/seed# cat prefix q suffix1 p suffix2 > part4b
root@VM:/home/seed# chmod +x part4b
root@VM:/home/seed# ./part4b
WARNING: malicious code
root@VM:/home/seed# 
```



```
/home/seed/part4.o - Bless
part4.o

0000102c 00 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 41 AAAAAAAAAAAAAAAAAAAAAAA
0000105a 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
00001071 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
00001088 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
0000109f 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
000010b6 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
000010cd 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
000010e4 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
000010fb 41 41 41 41 41 41 41 41 41 41 41 41 00 00 00 00 00 00 00 00 00 00 00 AAAAAAAAAAAA...........
00001112 00 00 00 00 00 00 00 00 00 00 00 00 00 00 41 41 41 41 41 41 41 41 41 ..............AAAAAAAAA
00001129 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
00001140 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
00001157 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA
0000116e 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAAAAAAA

  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: 4160 / 7871        Selection: None        INS
```

Task 4 – The first image is creating an array and a function to simulate a benign and a malicious code and creating a prefix for both of the MD5 hashes. The second image is creating the P and Q parts of the hash that will collide with a benign or a malicious code error. It's spitting the suffix of the hash into 2 pieces so that P can be inserted, and it creates the complete hashes. Once they are run, the hashes show the benign and malicious code errors. The third image shows the beginning of the X and Y arrays from the C code used to create the prefix and suffix,

Works Cited

For Password Dictionary List:
https://crackstation.net/crackstation-wordlist-password-cracking-dictionary.htm

For Hashing Algorithm:
https://github.com/yaoyao-cn/md5

For Checking/Creating the MD5 Hashes:
https://www.md5hashgenerator.com/

For Problem Definition, Brief Literature, MD5 Loop Image:
“Applied Cryptography: Protocols, Algorithms and Source Code in C, 20th Anniversary Edition”, by Bruce Schneier

For Brief Literature:
https://www.simplilearn.com/tutorials/cyber-security-tutorial/md5-algorithm#:~:text=MD5%20(Message%20Digest%20Method%205,means%20for%20digital%20signature%20verification

For Virtual Box:
https://www.virtualbox.org/wiki/Download_Old_Builds_6_0

For Ubuntu:
https://drive.google.com/file/d/12I8OO3PXHjUsf9vfjkAf7-I6bsixvMUa/view

For Collision Attack Lab:
https://seedsecuritylabs.org/Labs_20.04/Files/Crypto_MD5_Collision/Crypto_MD5_Collision.pdf

For Collision Attack:
https://www.youtube.com/watch?v=mGCVKLLjIns&ab_channel=KaityCodes