5850 Graduate Project: MD5 Collision Attack Lab

Ariannah Black

Nov 17, 2023

Introduction

The aim for SEED Labs' MD5 Collision Attack Lab is to educate students about the security of one-way hash functions and develop their understanding of how collision attacks work. Upon completion of the assignment, students should be more familiar with the one-way and collision-resistance properties of hash functions, developing and launching attacks on one-way hash functions, and the structure of the MD5 hash function specifically.

In order for students to reach these objectives, SEED Labs generated four tasks related to the MD5 hash and the function `md5collgen`, which would find MD5 collisions within seconds. Prior to the start of this lab, students are required to download the lab's source code. Since the tasks must be completed within a Linux environment, I had to set up a virtual machine for my lab environment; I used limactl. Following this, I was able to proceed with the lab.

Reference link to MD5 Collision Attack Lab description:
https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_MD5_Collision/

Tasks

The MD5 Collision Attack Lab is divided into four tasks; each is listed below alongside a walkthrough of my solutions and an explanation of my results.

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

Overview: Students will generate two different files which produce the same MD5 hash value and explore how `md5collgen` works, guided by discussion questions.

Methods: I first created an arbitrary text file (`prefix.txt`) to be used in later steps. Note that the length of the text file was 1043 bytes, which is not a multiple of 64. Following this, I used the below command to run `md5collgen` on my prefix file and generate two binary files (`out1.bin, out2.bin`).

(Command 1)

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

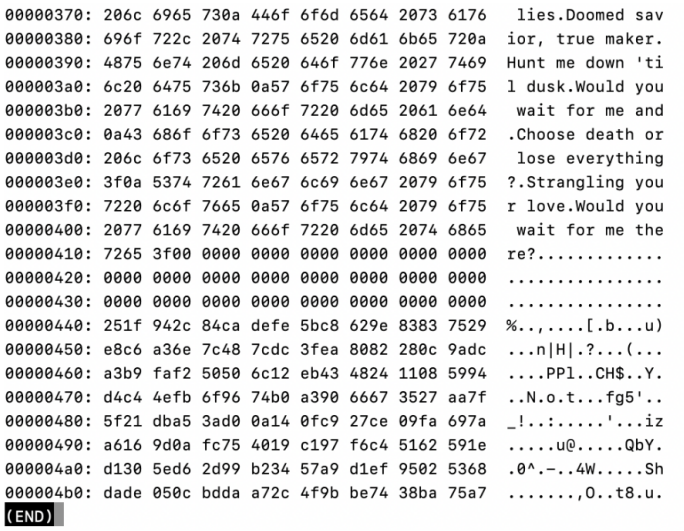
For the next portion of the task, students are to view their binary file outputs in a hex editor of their choice; the lab document suggests using `bless`, which is available within SEED Labs' VM. Since I was using my own VM for the project, and since my computer's OS is Unix-based, I used the below commands to output hex dumps of the binaries to console.

(Commands 2, 3)

```
xxd out1.bin | less
xxd out2.bin | less
```

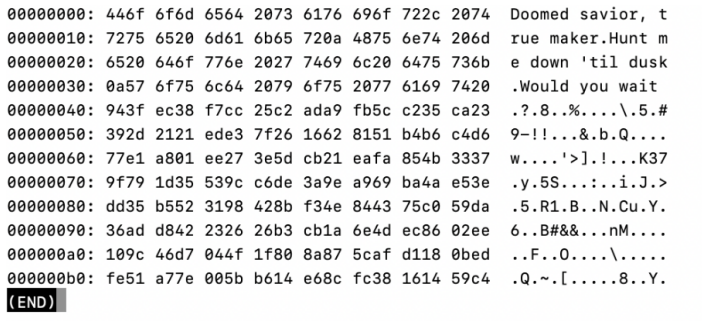
Below is a snippet of the end of the hex dump for `out1.bin`.

(Figure 1)



I next created another arbitrary text file, `prefix64.txt`, which was of length 64 bytes. Below is the hex dump for the first binary file generated from calling `md5collgen` on the new prefix.

(Figure 2)



Taking the two binary files sourced from `prefix64.txt`, I was able to find the differing bytes between them. The command and its output are displayed below.

(Figure 3)

```
(base) ariblack@owls Labsetup % cmp -l out1.bin out2.bin
    84  41 241
   110 113 313
   124 151 351
   148 102 302
   174  30 230
   175  13  12
   188  70 270
(base) ariblack@owls Labsetup %
```

Discussion (Q1): If the length of your prefix file is not multiple of 64, what is going to happen?

If the length of the prefix file is not divisible by 64, its last lines are padded in order to fill the remaining bytes. This is represented by the sequence of 0's in the hex in Figure 1, and with the sequence of ' . ' characters in the ASCII.

Discussion (Q2): Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.

If the length is already 64 bytes, then there will be no padding appended to the end of the prefix file. The hex dump thus looks more random than with the previous prefix.

Discussion (Q3): Are the (128 bytes) generated by `md5collgen` completely different for the two output files? Please identify all the bytes that are different.

The data is mostly the same, but a handful of bytes are different between the two files. From the output in Figure 3, we can see that the byte at offset 84 (byte 20) has value 41 in the first binary, and it is 241 in the second. So, at offsets 84, 110, 124, 148, 174, 175, and 188, we see different values for bytes 20, 46, 60, 84, 110, 111, and 124, respectively.

Task 2: Understanding MD5's Property

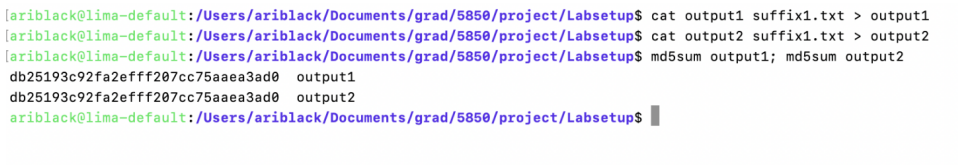
Overview: Students will demonstrate that, for MD5, the following property holds: if inputs M and N have the same hash, then adding the same suffix T to them will result in two outputs that contain the same hash value.

Methods: I first ran the `md5collgen` script on an arbitrary prefix text file and verified that the hash for both output files were the same.

(Figure 4)

I then created an arbitrary suffix text file and concatenated it to each of the binary output files. I then verified that, again, the MD5 hash values for the files were the same.

(Figure 5)



Discussion: From this exercise, I was able to better understand the property of MD5 previously presented in the lab document. I developed a deeper understanding of how the MD5 algorithm works through practical examples.

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

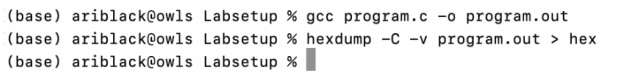
Overview: Students will generate two different executable files which produce the same MD5 hash. They will take a provided program and create two versions of it, such that the contents of their arrays are different, but the hash values of their executables are the same.

Methods: Utilizing the skeleton code provided in the lab document, I created a C program which initialized an unsigned char array of 0x41 (the hex value for ASCII character 'A') and printed its values.

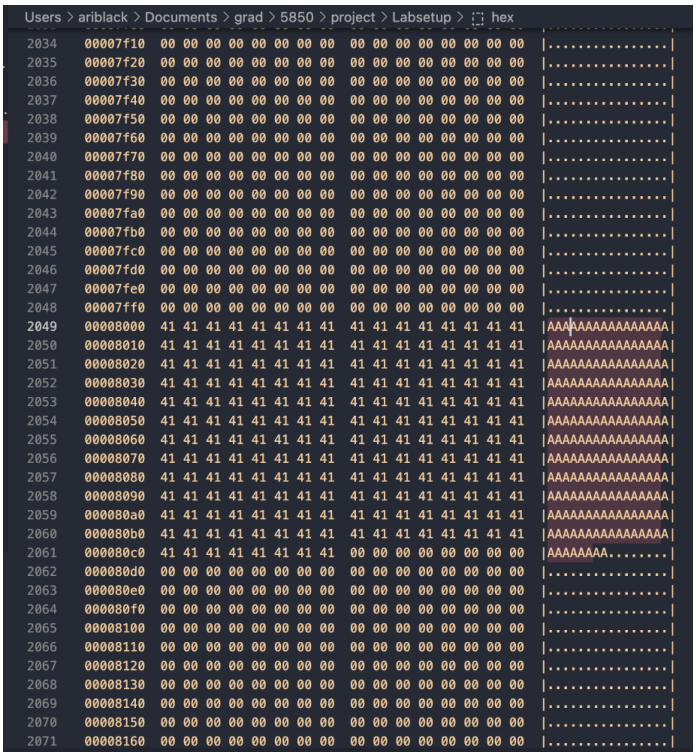
(Figure 6)

I compiled the program to get its binary executable, then stored its hex dump in a text file.

(Figure 6)



(Figure 7)



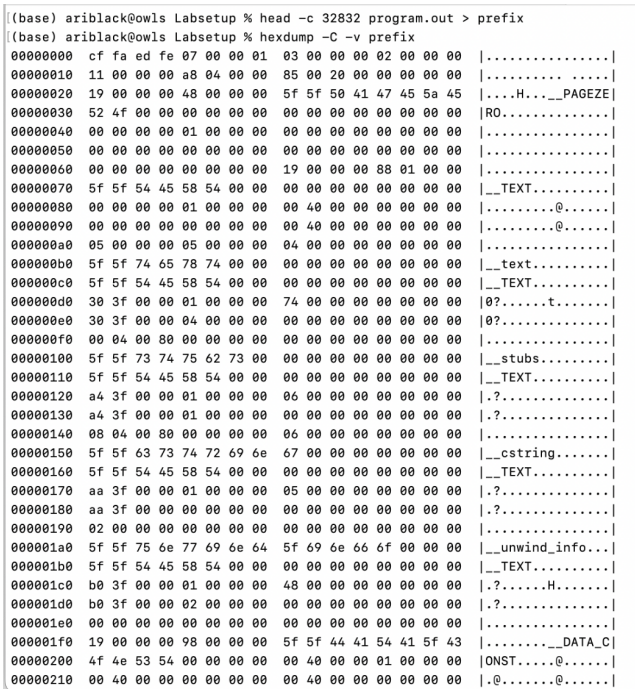
I used (⌘+F) to locate the contents of the array, which I knew would be a sequence of A's, and found that they began at 0x8000 (32768 decimal) and ended after 0x80c0 (32960). I noted that the end of the program reached 0xc118 (49432 decimal), which is a difference of 16472 bytes from the end of the array.

My next steps followed the guidelines of the property specified in the lab document:

```
MD5 (prefix || P || suffix) = MD5 (prefix || Q || suffix)
```

I generated my prefix text file by taking the first portion of the executable, up until just after the beginning of the array. The end of the array was at 0x80c0, so I ended the splice 128 bytes prior (as to make room for the 128-byte P/Q value, since the length of the overall executable must equal the length of the original executable), at 0x8040 (so after 32832 bytes). Note that the prefix has to be a multiple of 64.

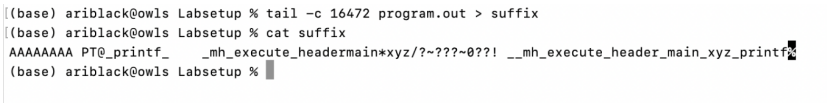
(Figure 8)



```
(base) ariblack@owls Labsetup % head -c 32832 program.out > prefix
(base) ariblack@owls Labsetup % hexdump -C -v prefix
00000000  cf fa ed fe 07 00 00 01  03 00 00 00 02 00 00 00  |................|
00000010  11 00 00 00 a8 04 00 00  85 00 20 00 00 00 00 00  |.......... .....|
00000020  19 00 00 00 48 00 00 00  5f 5f 50 41 47 45 5a 45  |....H..__PAGEZE|
00000030  52 4f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |RO..............|
00000040  00 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00  19 00 00 00 88 01 00 00  |................|
00000070  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00  |__TEXT..........|
00000080  00 00 00 00 01 00 00 00  00 40 00 00 00 00 00 00  |.........@......|
00000090  00 00 00 00 00 00 00 00  00 40 00 00 00 00 00 00  |.........@......|
000000a0  05 00 00 00 05 00 00 00  04 00 00 00 00 00 00 00  |................|
000000b0  5f 5f 74 65 78 74 00 00  00 00 00 00 00 00 00 00  |__text..........|
000000c0  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00  |__TEXT..........|
000000d0  30 3f 00 00 01 00 00 00  74 00 00 00 00 00 00 00  |0?......t.......|
000000e0  30 3f 00 00 04 00 00 00  00 00 00 00 00 00 00 00  |0?..............|
000000f0  00 04 00 80 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000100  5f 5f 73 74 75 62 73 00  00 00 00 00 00 00 00 00  |__stubs.........|
00000110  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00  |__TEXT..........|
00000120  a4 3f 00 00 01 00 00 00  06 00 00 00 00 00 00 00  |.?..............|
00000130  a4 3f 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |.?..............|
00000140  08 04 00 80 00 00 00 00  06 00 00 00 00 00 00 00  |................|
00000150  5f 5f 63 73 74 72 69 6e  67 00 00 00 00 00 00 00  |__cstring.......|
00000160  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00  |__TEXT..........|
00000170  aa 3f 00 00 01 00 00 00  05 00 00 00 00 00 00 00  |.?..............|
00000180  aa 3f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.?..............|
00000190  02 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001a0  5f 5f 75 6e 77 69 6e 64  5f 69 6e 66 6f 00 00 00  |__unwind_info...|
000001b0  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00  |__TEXT..........|
000001c0  b0 3f 00 00 01 00 00 00  48 00 00 00 00 00 00 00  |.?......H.......|
000001d0  b0 3f 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |.?..............|
000001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001f0  19 00 00 00 98 00 00 00  5f 5f 44 41 54 41 5f 43  |........__DATA_C|
00000200  4f 4e 53 54 00 00 00 00  00 40 00 00 01 00 00 00  |ONST.....@......|
00000210  00 40 00 00 00 00 00 00  00 40 00 00 00 00 00 00  |.@.......@......|
```

The length of my suffix would be the length of the remainder of the executable, after the array (so 16472 bytes). With this information, I generated my suffix (which should also be a multiple of 64).

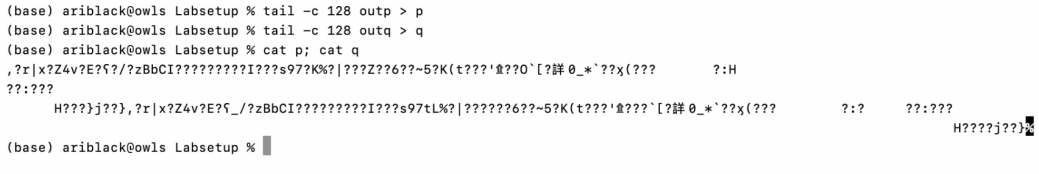
(Figure 9)



```
(base) ariblack@owls Labsetup % tail -c 16472 program.out > suffix
(base) ariblack@owls Labsetup % cat suffix
AAAAAAAA PT@_printf_    _mh_execute_headermain*xyz/?~???~0??! __mh_execute_header_main_xyz_printf
(base) ariblack@owls Labsetup %
```

To get my P and Q values, I ran the `md5collgen` script on the prefix and extracted 128 bits (`p, q`) from each of the resulting binaries (`outp, outq`).

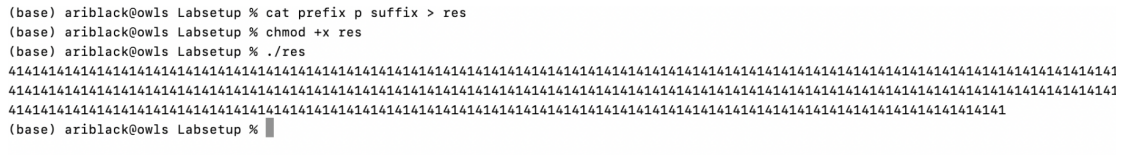
(Figure 10)



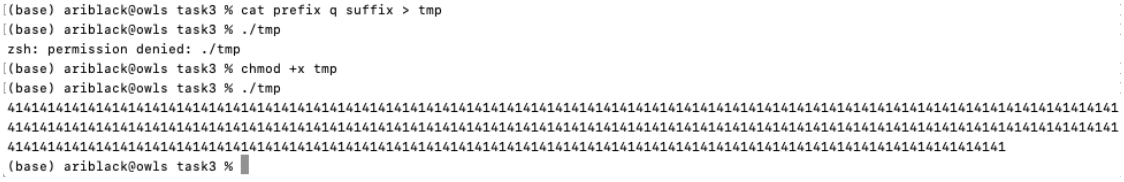
```
(base) ariblack@owls Labsetup % tail -c 128 outp > p
(base) ariblack@owls Labsetup % tail -c 128 outq > q
(base) ariblack@owls Labsetup % cat p; cat q
,?r|x?Z4v?E?ʕ?/?zBbCI?????????I???s97?K%?|???Z??6??~5?K(t???'𦰡??O`[?詳0_*`??ɣ(???       ?:H
??:???
      H???}j??},?r|x?Z4v?E?ʕ_/?zBbCI??????????I???s97tL%?|??????6??~5?K(t???'𦰡???`[?詳0_*`??ɣ(???        ?:?     ??:???
                                                                                                    H????j??}
(base) ariblack@owls Labsetup %
```

I then built my two executables by stitching the pieces together (prefix || p || suffix for one, and prefix || q || suffix for the other) and changing execution permissions for the files. After verifying that the hash for both programs were correct, I knew that I had successfully completed Task 3.

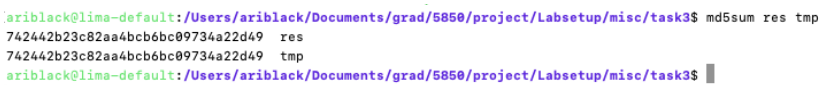
(Figure 11)



(Figure 12)



(Figure 13)



Discussion: This exercise gave me significantly deeper insight regarding finding collisions in one-way hash functions. The process of splicing and re-piecing together different bits of data, and especially the process of locating the specific bytes needed for splicing, proved to be a bit more difficult but more academically fulfilling than I expected.

Task 4: Making the Two Programs Behave Differently

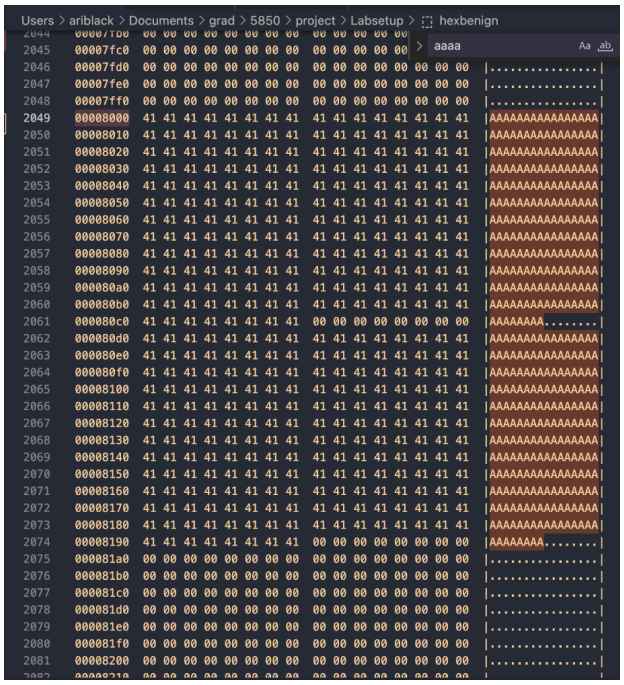
Overview: Students will generate two different programs which produce the same MD5 hash. Execute an attack demonstrating that it is possible to obtain a certificate (hash value) which applies to two different sets of instructions.

Methods: I used the provided skeleton pseudo code as a base to develop my main C program, shown below. My program initializes two unsigned char arrays, both filled with the value 0x41 (ASCII character 'A'). It then includes a check: if the two arrays are the same, then it will execute a set of benign instructions; otherwise, it will execute its malicious instructions.

(Figure 14)

I compiled the code to get its binary executable, then stored its hex dump in another text file, shown below. From there, I found the locations of the two arrays. The first began and ended at the same locations as the array from the previous task (0x8000/32768 and 0x80c0/32960, respectively). The second began and ended at 0x80d0/32976 and 0x8190/33168.

(Figure 15)



Similar to in Task 3, the lab document presents another approach to generating a hash collision; the structure of the executables I generated aligned with the following guiding principle:

```
MD5(prefix || p || middle || p || suffix)
= MD5(prefix || q || middle || p || suffix)
```

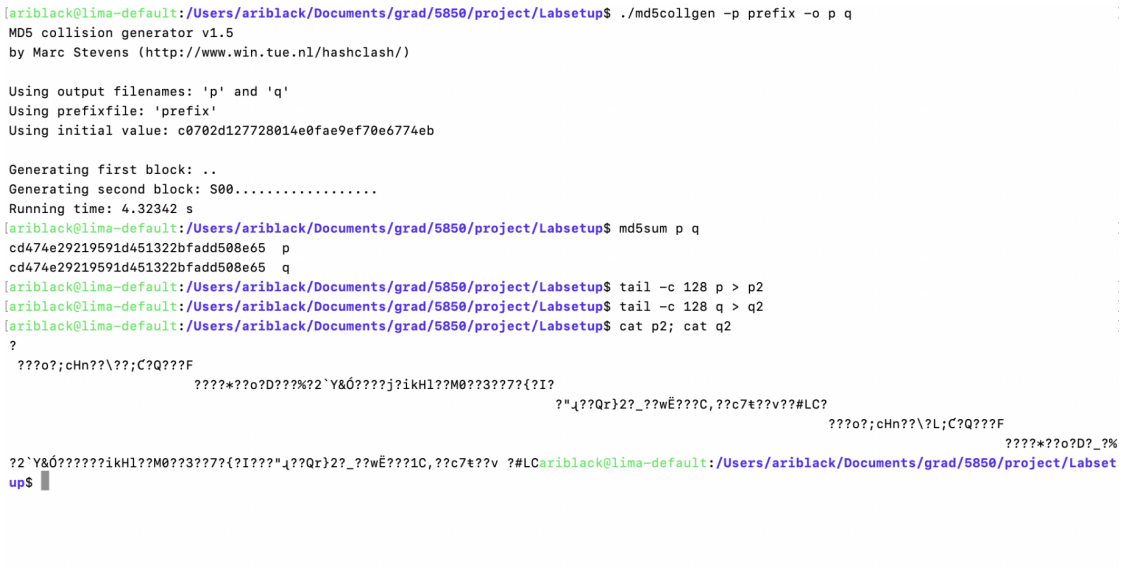
To generate my prefix, I spliced the executable from its start to byte 32832, 64 bytes after the beginning of the first array. For the suffix, I first took all bytes after 32961 (the end of our first splice + 128, the length of p/q), then obtained the final 209 bytes from that selection. I assigned to the middle value the first 80 bytes immediately after 32961.

(Figure 16)

```
(base) ariblack@owls Labsetup % head -c 32832 benign.out > prefix
(base) ariblack@owls Labsetup % tail -c +32961 benign.out > tmp
(base) ariblack@owls Labsetup % tail -c +209 tmp > suffix
(base) ariblack@owls Labsetup % head -c 80 tmp > tmp2
```

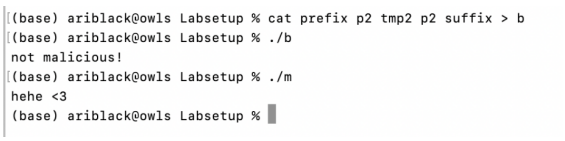
To generate the p and q values, I called the `md5collgen` script on the prefix, verified that their hashes were the same, and extracted 128 bytes from both p and q.

(Figure 17)

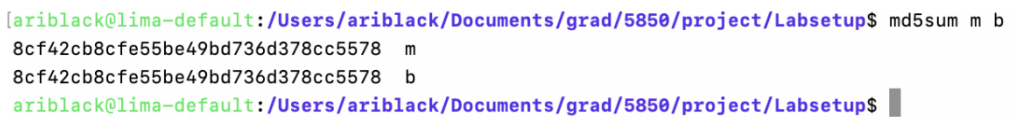


I was then able to concatenate the parts to create my executables. My benign program had the structure `prefix || p || middle || p || suffix`, so both arrays within the program were the same, leading its control flow to execute the benign instructions. The malicious program had a slightly different structure, `prefix || q || middle || p || suffix`, where the one array is modified, leading the control flow to execute the malicious instructions. After adding execution permissions for both programs, I executed both as shown below (where `b` is the name of the benign program, and `m` is the malicious one) and verified that their hashes remained the same.

(Figure 18)

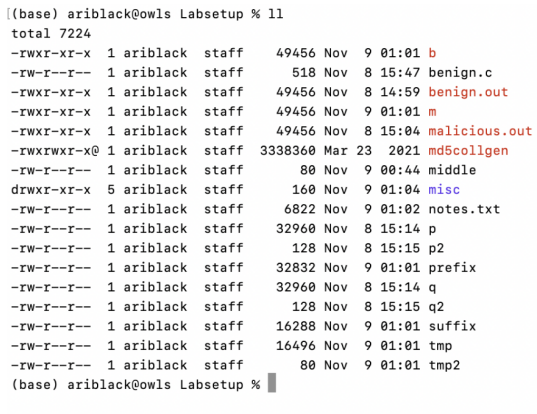


(Figure 19)



The lengths of the files I used in my program concatenation were (in bytes) 32832, 128, 80, 128, and 16288 for the prefix, p (labeled p2 below), middle (labeled tmp2 below), q (labeled q2 below), and suffix splice sections, respectively.

(Figure 20)



Discussion: This exercise was the most challenging task within the lab for me as, although I had a good general idea of how I planned to splice the different pieces together (very similarly to in Task 3), I struggled with finding the precise number of bytes that each piece should hold. I spent a good portion of time experimenting with different values for the `head` and `tail` commands, saving the resulting hex dump to a file, and observing the architecture of the output in order to see precisely why I was receiving the errors I found; eventually, I realized that my suffix and middle splices were off my 1 byte (when fixed, my scripts worked as expected).

Discussion

Overall, I believe that this lab was successful in its goal to teach me about one-way hash function vulnerabilities and about collision attacks, specifically using MD5. While I previously had a good general understanding of how collisions work and their impact on security, stepping through each of the four tasks above allowed me to understand the security risks at a more sophisticated level.

Upon completion of the lab, I can conclude that the MD5 hash function's security is severely compromised, and that it generally should no longer be used in favor of other, more secure hash functions.