|  |  |  |
| --- | --- | --- |
| Luke Pepin | CSE 4400 - P1 | 2/21/2025 |

**Introduction**

During this lab, the MD5 one-way hash function is tested against collision attacks. Collision attacks exploit weaknesses in a hash function’s collision-resistance property by finding two different inputs that produce the same hash. Exploring these attacks against MD5 highlights the importance of secure hash functions in maintaining data integrity and security in real-world applications.

**Lab Tasks:**

The MD5 Collision Attack Lab is comprised into four tasks; listed below are my solutions and explanation of results.

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

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Creation of a prefix.txt file | echo "luke pepin - CSE 4400 P1" > prefix.txt |
| Confirmation of prefix.txt file size | stat –c%s “prefix.txt” |
| Generation of Hash Collision Files | md5collgen -q -p prefix.txt -o O1.bin O2.bin |
| Comparison of File Contents | diff O1.bin O2.bin |
| Comparison of File Hashes | md5sum O1.bin O2.bin |
| Binary Review of File Content | bless O1.bin  bless O2.bin |

Explanation:

The generation of two different files which have the same MD5 has value begins with the creation of a prefix.txt file. This file serves as the beginning of both files with the hash collision.

Next, the prefix file length is confirmed (multiple tests were run with differing byte lengths which will be discussed in the questions section). It this case the length is not a multiple of 64.

Following is the md5collgen, collision generator command which has the following parameters: -q: results in less output text (optional and doesn’t affect lab), -p prefix.txt: declares each hash collision file is to start with the content from prefix, -o O1.bin O2.bin: States the output files (ones with the same MD5 hash) will be called O1.bin and O2.bin.

Comparison of file contents has been completed with the diff O1.bin O2.bin command which resulted in the output “Binary files O1.bin and O2.bin differ”. Confirming the file content is different.

The md5sum command is used to retrieve the hash of both O1.bin and O2.bin displaying that they are in fact the same

“0802c6063127fa58c1336b0923ac8ce9 O1.bin

0802c6063127fa58c1336b0923ac8ce9 O2.bin”.

Lastly, a binary review of the output files was conducted to determine which bytes had alterations.

Questions from Instructions:

Question 1. If the length of your prefix file is not multiple of 64, what is going to happen?

If the length of the prefix is not a multiple of 64 bytes the MD5 Collision generation tool would automatically pad the prefix fix with added bytes to make the length the proper size.

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

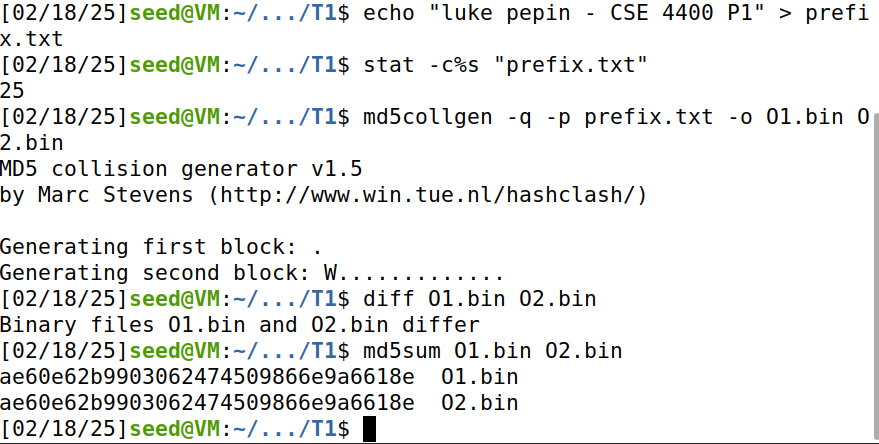
No padding was added to the end of the output files. Here is the example: A screenshot of a computer

AI-generated content may be incorrect.

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.

If you scroll down to the last screenshot all the highlighted bytes are changed between the output files.

Screenshots:



LP\_P1\_T1\_1.png: Command lines for completion of Task 1; Creation of prefix.txt, Confirmation of file size of prefix.txt, File generation with the same MD5 hash, Verification of different files, MD5 hash check and finally binary review of output files.

A screenshot of a computer screen

AI-generated content may be incorrect.LP\_P1\_T1\_2.png: Bless review of O1.bin, note in the second to last line the third character is “8”.

A screenshot of a computer screen

AI-generated content may be incorrect.

LP\_P1\_T1\_3.png: Bless review of O2.bin, note in the second to last line the third character is “.”.

A screenshot of a computer

AI-generated content may be incorrect.

LP\_P1\_T1\_4.png: Highlighted bytes that differ from O1.bin and O2.bin

**Task 2: Understanding MD5’s Property**

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Create a prefix file | echo "luke pepin - CSE 4400 P1 T1" > prefix.txt |
| Generation of two output files with same hash value | md5collgen -q -p prefix.txt -o 1T.bin 2T.bin |
| Concatenate a suffix to both files | echo “CSE 4400 P1 T2 ENDING” > suffix.txt  cat 1T.bin suffix.txt > 1TS  cat 2T.bin suffix.txt > 2TS |
| Verify the Hashes of the files with suffixes | md5sum 1TS 2TS |

Explanation:

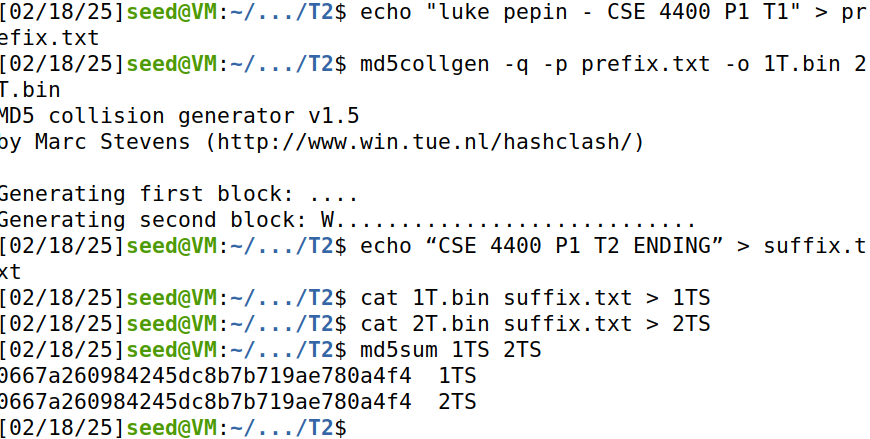
The first two steps from this lab are very similar to the first task; first the same prefix file is created and two files with the same hash are generated with the prefix.

The important element of this task is the creation of and the concatenation of a suffix file unto the hash collision files.

The hashes of the output files are confirmed afterwards to see that they remain the same as each other.

The output of both commands was the same despite the new suffix, the files still have the same hash value. Demonstrates that appending the same suffix on two files with the same hash results in outputs with the same hash.

Screenshot:



LP\_P1\_T2\_1.png: Command lines for completion of Task 2; Creation of suffix.txt (not shown), appendage of suffix file to output files, confirmation of commands (ls & bless sout1.bin) and verification of MD5 hashes of new files.

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

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Create an executable file | nano T3a.c  gcc T3a.c –o T3a.o |
| Find the prefix/128 byte offset border | bless T3a.o |
| Separate the executable prefix | head -c 12352 T3a.o > prefix |
| Generate two files with the same MD5 hash | md5collgen -p prefix -o 3T.bin 4T.bin |
| Separate the executable suffix | tail -c +12480 T3a.o > suffix |
| Save the P and Q values | tail -c 128 3T.bin > p  tail -c 128 4T.bin > q |
| Append the array contents and the suffix contents to the hashed files | cat prefix p suffix > T3f  cat prefix q suffix > T4f |
| Verify the values of the hash | md5sum T3f T4f |
| Execution of files | chmod +x T3f T4f  T3f  T4f |

Explanation

To showcase that contents of executable files can be altered while having the same hash first what is needed is a executable file. T3a.c is a executable file with a long array of letter A.

Next under review of that file with blessing it can be determined that towards the beginning of the array in a number divisible by 64 bytes to prevent padding is 12352 bytes in the executable.

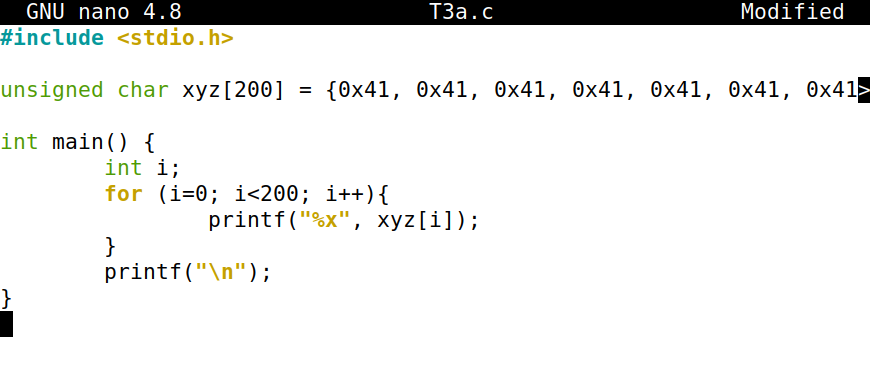
The acquisition of this figure allows us to create a prefix which is interrupted at the center of the array.

Since the prefix file cuts the center of the array when generating two files with the same hash elements of the array may be altered.

The executable is then rebuilt with its original prefix and suffix, but they are built with the hash collision array elements.

The values of the hash are confirmed to remain the same and both files are executed showcasing different array values.

Screenshots:

**T3a.c:**

A screenshot of a computer

AI-generated content may be incorrect.

LP\_P1\_T3\_1.png: Bless T3a.o output and the search to find the A array, once found located the offset and chose a number that is divisible by 64 within the array to define the prefix/128-byte region border.

A computer screen shot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

LP\_P1\_T3\_2.png: Screenshot of the suffix and 128-byte region creations along with the concatenation of the executables. I was able to confirm the hashes remain the same despite different variable with a md5sum check of both executables. Note: on the 3 to last line all the way on the right side 782 != 5e3.

**Task 4: Making the Two Programs Behave Differently**

|  |  |
| --- | --- |
| **Task** | **Commands** |
| Creation of an executable that changes if hash is altered. | nano T4.c  gcc T4.c –o T4.o |
| Find the start of 128-byte regions of the arrays in the executable | bless T4.o |
| Separate the executable prefix | head -c 12352 T4.o > prefix |
| Generate two files with the same MD5 hash | md5collgen -p prefix -o 5T.bin 6T.bin |
| Save the p and q values | tail -c 128 5T.bin > p  tail -c 128 6T.bin > q |
| Separate the executable suffixes | tail -c +12481 T4.o > suffix  head -c 96 suffix > S1  tail -c +223 suffix > S2 |
| Append the array contents and the suffix contents to the hashed files | cat prefix p S1 p S2 > T5f  cat prefix q S1 p S2 > T6f |
| Verify the values of the hash | md5sum T5f T6f |
| Execution of files | chmod +x T5f T6f  T5f  T6f |

Explanation:

First, an executable is created from the code in T4.c, which contains two long arrays with the same value. The code checks if these arrays are the same and prints "Good Ending" if they are, otherwise, it prints "Bad Ending". This code is compiled into an executable named T4.o.

Next, the executable T4.o is examined to find the start of the 128-byte regions of the arrays. The first byte divisible by 64 is found at 12352 bytes into the executable. Everything before this point is saved as the prefix.

Using the prefix, two files (5T.bin and 6T.bin) with the same MD5 hash are generated. The last 128 bytes of the hash-colliding files are saved into p and q. These values will be used for swapping.

The suffix of the executable starts at 12481 bytes, which is 128 bytes + 1 past where the prefix stops. This suffix is divided into two sections: S1 (the area between the 128-byte regions of p and q) and S2 (the area after the second 128-byte region).

The files are concatenated in two different forms:

* T5f: prefix | p | S1 | p | S2
* T6f: prefix | q | S1 | p | S2

The MD5 hashes of both files (T5f and T6f) are checked to ensure they are equal. The files are then made executable and run. Despite having the same hash, they produce different outputs ("Good Ending" or "Bad Ending").

Screenshots:

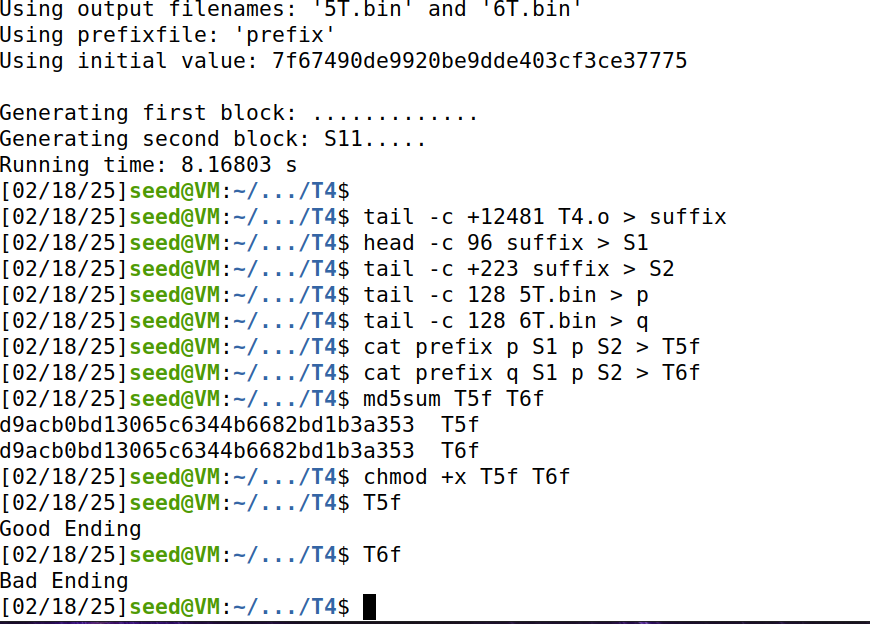
**T4.c:**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



LP\_P1\_T4\_1.png: As you can see the md5sum of T5f and T6f remain the same yet their output are different.

**Conclusion:**

In conclusion, this lab effectively demonstrates the vulnerabilities of the MD5 hash function to collision attacks. By generating different files with identical MD5 hashes and observing their behavior, we highlight the critical need for more secure hash functions in protecting data integrity. The experiments underscore the importance of understanding and mitigating such weaknesses in real-world applications to ensure robust security measures.