Lab 9: MD5 Collision Attack Lab (a SEED Lab)

Shivali Mate

I. Introduction

THE purpose of this lab was to explore the vulnerabilities of the MD5 hashing algorithm by practically demonstrating how two different files can share the same hash value. Although MD5 was once widely used for integrity verification, researchers such as Xiaoyun Wang et al. [2] and Marc Stevens [3] showed that it is susceptible to collision attacks that compromise its security. Through this experiment, the student gained a deeper understanding of the concept of collision resistance and how its failure can enable the creation of distinct programs that appear identical when verified using MD5.

II. Methodology

The lab used md5collgen to generate colliding files, verified identical hashes, demonstrated MD5's concatenation property, and created two executables with a 64-byte aligned collision block that produced the same hash but different behaviors.

A. Task 2.1

Two files with identical MD5 hashes were generated using md5collgen. Random prefix files, prefix.bin and prefix64.bin were created with head -c and tested with the following commands as shown in Fig. 1.

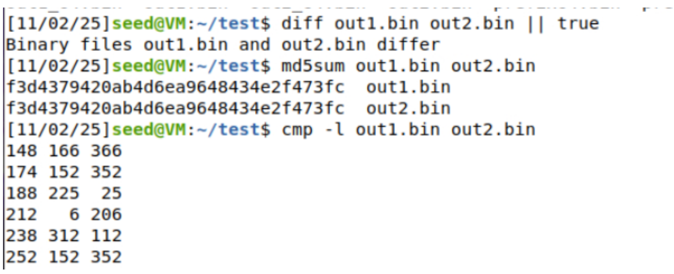

Fig. 1. Two different binary files sharing the same MD5 hash

The questions are answered based on the observations after running the commands.

1) If the length of your prefix file is not multiple of 64, what happens?
   Even when the prefix length of 70 bytes which is not a multiple of 64, md5collgen still successfully generates a collision. The tool automatically applies MD5 padding to align the data to 64-byte blocks internally, so no error occurs, the collision region is simply shifted to start in the next padded block.
2) Create a prefix file with exactly 64 bytes, and run the collision tool again. What happens?
   When the prefix is exactly 64 bytes i.e a full MD5 block, md5collgen again generates two colliding outputs. Because the prefix already aligns with MD5's block size, the collision blocks start immediately after the prefix without extra padding. The process runs a bit faster and produces valid collisions just like before.
3) Are the 128 bytes generated by md5collgen completely different for the two output files?
   Identify all bytes that are different. No, the 128-byte collision blocks are not completely different; only specific bytes differ between out1.bin and out2.bin. Your cmp output shows the differing byte offsets and values:

```
148 166 366
174 152 352
188 225 25
212 6   206
238 312 112
252 152 352
```

   These lines indicate exactly where the two files differ within the 128-byte region. Despite these few differences, both files yield the same MD5 hash, demonstrating the collision.

B. Task 2.2

The concatenation property of MD5 was demonstrated by appending a common suffix to both colliding files:

```
cat out1.bin suffix.txt >
    out1_concat.bin
cat out2.bin suffix.txt >
    out2_concat.bin
```

As shown in Fig. 2, both concatenated files maintained identical hashes, confirming the property:

```
MD5(M||T) = MD5(N||T)
when MD5(M) = MD5(N)
```

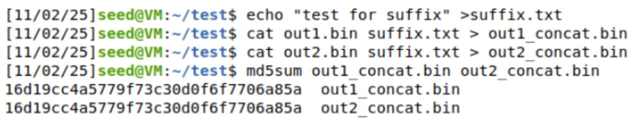

Fig. 2. Identical hashes after concatenation of files

C. Task 2.3

A C program containing a large xyz array filled with repeated 0x41 bytes was created and compiled. The resulting executable was examined using bless to locate the long 0x41 region as seen in Fig. 3 and then split into three parts, a prefix, a 128-byte collision region, and a suffix, using head and tail. The split point was chosen by identifying the byte offset where the replacement should start and selecting a prefix length that places the 128-byte block squarely inside the xyz data, so the replacement affects only the data region and not

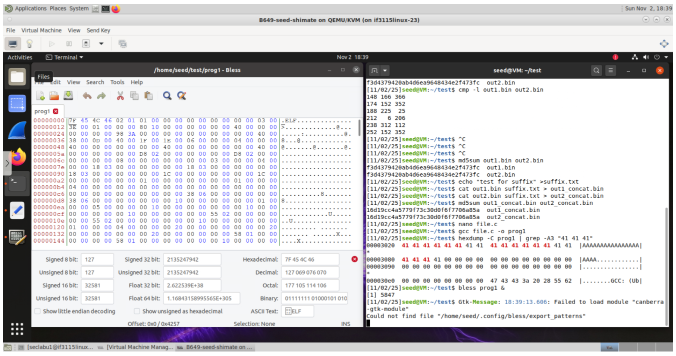

Fig. 3. Bless showing 0x41 bytes of data.

ELF headers or code. The prefix and suffix were extracted with commands

```
head -c <prefix_len> prog1
    > prefix
tail -c +$((<prefix_len>+128+1))
    prog1 > suffix
```

Colliding 128-byte blocks were generated with md5collgen and then inserted by concatenation to produce two modified executables. The modified binaries were inspected in bless and tested. Fig. 4 shows the md5sum verification and identical hashes alongside a comparison of the two modified program files and their differing runtime outputs.

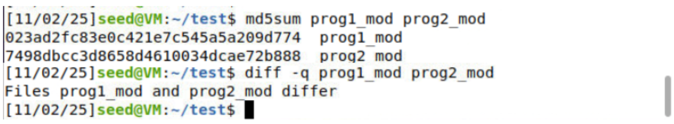

Fig. 4. Same MD5 hash with different data.

D. Task 2.4

In the final task, the program was modified to include a 128-byte array that determined which branch of code executed. The array was aligned to a 64-byte boundary so the collision block could be inserted cleanly without affecting nearby data. This alignment was essential because MD5 processes data in 64-byte blocks; aligning the prefix ensured that the generated collision block matched the algorithm's internal structure and produced valid identical hashes.

After compiling the program with all bytes set to 0xAA, the start of the array was located using hexdump, and a prefix length that was a multiple of 64 was selected. The prefix and suffix were extracted, and md5collgen was used to generate two colliding 128-byte blocks.

These were merged back with the suffix to form two executables. A one-byte change in one file flipped the checksum parity, causing one program to print Benign path while the other displayed Malicious path witch was obtained by using:

```
cat suffix >> agen
cat suffix >> bgen
```

Fig 5 show the aligned region, identical MD5 hashes, and the differing runtime results confirming the success of the collision.

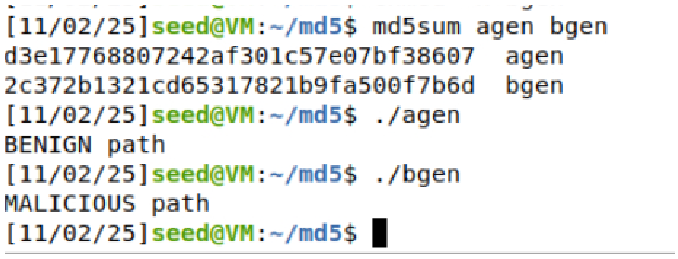

Fig. 5. Malicious and Benign files with same MD5 hash

III. Tools and Learning Experience

The lab was conducted in the SEED Ubuntu 20.04 VM environment using command-line tools such as md5collgen, diff, cmp, hexdump, and the hex editor Bless. Compilers like gcc and utilities like head and tail were essential for splitting and reconstructing binaries. Hands-on experience is gained with binary editing, file alignment, and the inner workings of hash-based integrity verification. This practical exposure highlighted the real-world implications of weak hash algorithms that remain in legacy systems.

IV. Findings and Discussion

The experiments clearly demonstrated MD5's vulnerability to collision attacks. In each case, distinct files or executables generated the same MD5 hash while differing in binary content or runtime behavior. Fig 5 especially emphasized how attackers could submit a benign program for certification and later substitute a malicious one with the same MD5 hash, precisely the threat outlined in the SEED Lab manual [1] and studies by Wang et al. [2]. The work of Stevens et al. [4] on SHA-1's "SHAttered" collision further confirmed that such weaknesses persist even in newer algorithms, underscoring the need for cryptographically secure alternatives such as SHA-256.

V. Conclusion

The MD5 Collision Attack Lab effectively illustrated how theoretical vulnerabilities translate into real exploits. By generating and embedding colliding blocks into programs, it was demonstrated that hash equality does not guarantee content integrity. The exercise strengthened understanding of the collision-resistance property, highlighted the importance of cryptographically strong hash functions, and showed how weak hashing can undermine software verification and digital certification processes.

References

[1] Wenliang Du. SEED Labs – MD5 Collision Attack Lab. Syracuse University, 2018.
[2] Xiaoyun Wang, et al. "Collisions for Hash Functions MD5, SHA-0, and SHA-1." CRYPTO Rump Session, 2004.
[3] Marc Stevens. On Collisions for MD5. Master's Thesis, Eindhoven University of Technology, 2007.
[4] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The First Collision for Full SHA-1 (SHAttered Attack). CWI Amsterdam and Google Research, 2017.