Assignment 5 Report
MD5 Collision Attack Lab
Demarcus Simmons
University of West Florida: CNT4403
Nov 11, 2023

Table of Contents

Introduction

For this assignment, I will be gaining hands-on experience with one-way hash functions. The objective of this lab is to understand the impact of collision attacks, and see what damages can be caused if a widely-used one-way hash function's collision-resistance property is broken. To achieve this goal, I'll need to launch actual collision attacks against the MD5 hash function.

Materials and Methods

The materials used is this assignment are:

- C programming language
- Python programming language
- VMware (Unbuntu)

The assignment will be completed in a series of tasks, which I will complete and document in order.

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

In this task, I will generate two different files with the same MD5 hash values. Using the collision generator program provided, I was able to execute it with the prefix.txt file and create two files (out1.bin & out2.bin) that differ as you can see in the following result.

```
[11/14/23]seed@VM:~/.../Labsetup$ 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: 4425255e13754315d429164205f04150

Generating first block: ........
Generating second block: S01..
Running time: 5.94642 s
[11/14/23]seed@VM:~/.../Labsetup$ diff out1.bin out2.bin
Binary files out1.bin and out2.bin differ
```

Even though the files differ you can see that they still produce the same MD5 Hash in the following screenshot:

```
Binary files out1.bin and out2.bin differ
[11/14/23]seed@VM:~/.../Labsetup$
[11/14/23]seed@VM:~/.../Labsetup$ md5sum out1.bin
7939f7c20bf7746fb5d29edf2fa12901  out1.bin
[11/14/23]seed@VM:~/.../Labsetup$ md5sum out2.bin
7939f7c20bf7746fb5d29edf2fa12901  out2.bin
[11/14/23]seed@VM:~/.../Labsetup$
```

Following are my answers the following questions for this task:

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 file isn't a multiple of 64 bytes, padding will be added to make it a multiple of 64 before applying the collision-generating tool.

Question 2. Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what Happens.
After running the collision tool again on a prefix file now exactly 64 bytes long the following was the result.

```
[11/26/23]seed@VM:~/.../Labsetup$ ./md5collgen -p prefix.txt -o out1.bin out2.bi
n
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: cc6d48f9783baf018e37c87de09a3371

Generating first block: ...
Generating second block: S11.....................................................
.....................................
Running time: 14.5937 s
```

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.
After running diff on the produced output files, I get the following result:

```
[11/26/23]seed@VM:~/.../Labsetup$ diff out1.bin out2.bin
2c2
< Z����3.@a9ǵh@a@6BP.���q:Q���:(
U@N>:@@Ub@T7`���Zd@@Eb@ @@.@@$@d/@)@@.����_@A@ǵ @>*@k@@@$        .A@I����@g@@@
\ No newline at end of file
---
> Z����3.@a9ǵh@ah6BP.���q:Q���:(
U@N>:@@Ub@T7����Zd@@Eb@ @@.@@$@d/@)@@@.����_@A@ǵ        @>*@k@@@@$       .A@I@@@@
@g@@@
\ No newline at end of file
```

However the hash values generated for each output file is the same as you can see in the following screenshot:

```
[11/26/23]seed@VM:~/.../Labsetup$ md5sum out1.bin
d9885e67b63b2410654b2fb8658fff40  out1.bin
[11/26/23]seed@VM:~/.../Labsetup$ md5sum out2.bin
d9885e67b63b2410654b2fb8658fff40  out2.bin
```

Task 2: Understanding MD5's Property

In this task, I will try to understand some of the properties of the MD5 algorithm. The MD5 hash algorithm is built using a construction called Merkle-Damgard construction. . The core of the MD5 algorithm is a compression function, which takes two inputs, a 64-byte data block and the outcome of the previous iteration. The compression function produces a 128-bit IV, which stands for "Initialization Vector"; this output is then fed into the next iteration. The output from the last iteration is the final hash value.

Based on how MD5 works, we can derive the following property of the MD5 algorithm: Given two inputs M and N, if MD5(M) = MD5(N), i.e., the MD5 hashes of M and N are the same, then for any input T, MD5(M || T) = MD5(N || T), where || represents concatenation.

To demonstrate, I took the files out1.bin (M) and out2.bin (N), which we already know have the same hash value, and appended a suffix.txt file (T) to each one to produce M || T and N || T. In the image below, you can see I was able to produce two different files from the original out1.bin and out2.bin files concatenated with an arbitrary suffix file.

```
[11/26/23]seed@VM:~/.../Labsetup$ cat out1.bin suffix.txt > out1_long.bin
[11/26/23]seed@VM:~/.../Labsetup$ cat out2.bin suffix.txt > out2_long.bin
[11/26/23]seed@VM:~/.../Labsetup$ diff out1_long.bin out2_long.bin
2c2
< Z����3.@a9ǵh@a@6BP.���q:Q���:(
U@N>:@@Ub@T7`���Zd@@Eb@ @@.@@$@d/@)@@.����_@A@ǵ @>*@k@@@$        .A@I@@@@g@@@A s
uffix file with data.
---
> Z����3.@a9ǵh@ah6BP.���q:Q���:(
U@N>:@@Ub@T7����Zd@@Eb@ @@.@@$@d/@)@@@.����_@A@ǵ        @>*@k@@@@$       .A@I@@@@
@g@@@A suffix file with data.
```

After applying the hash function, they both produced the same md5 hash value, as you can see in the following screenshot:

```
[11/26/23]seed@VM:~/.../Labsetup$ md5sum out1_long.bin
bb668fe30704e7a2bae4af3b061e84ba  out1_long.bin
[11/26/23]seed@VM:~/.../Labsetup$ md5sum out2_long.bin
bb668fe30704e7a2bae4af3b061e84ba  out2_long.bin
```

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

In this task, given a C program, my job is to create two different versions of this program, such that the contents of their xyz arrays are different, but the hash values of the executables are the same.
For a given program executable, it is possible to parse the file into three parts: a prefix, a 128-byte region, and suffix. The length of the prefix has to be a multiple of 64.
Using the md5collgen program, I was able to generate two outputs with the same hash using the prefix. The program I'm using in this task will simply print out a list of 200 ints, which initially holds 200 hex values representing the character A (0x41). To find the offset of the array in the executable I used the xxd utility program and here's the output:

```
[11/26/23]seed@VM:~/.../Labsetup$ xxd -c 16 a.out | grep AA
00003020: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00003030: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00003040: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00003050: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00003060: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00003070: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00003080: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00003090: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
000030a0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
000030b0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
000030c0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
000030d0: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
000030e0: 4141 4141 4141 4141 4743 433a 2028 5562  AAAAAAAAGCC: (Ub
```

From the results you can see that the array starts at an offset of 0x3020 which is 12320 in decimal. This is also unfortunate because this isn't a multiple of 64, which means I'll have to do some manual labor trying to find a multiple. However, I was able to create a simple python script to speed up the process and found 12352 to be a multiple of 64. I can now use this value to start parsing the prefix and suffix of the executable using the following commands:

```
[11/26/23]seed@VM:~/.../Labsetup$ head -c 12352 a.out > prefix
[11/26/23]seed@VM:~/.../Labsetup$ tail -c +12480 a.out > suffix
```

Using the md5collgen program with the newly created prefix file, I can create two new output files that can be used to create two 128 byte files P and Q that have the same hash value.

```
[11/26/23]seed@VM:~/.../Labsetup$ ./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: 1f5b5e383ed002dfca8a90a1dc1bb8dc

Generating first block: .........
Generating second block: W.......
Running time: 14.8657 s
[11/26/23]seed@VM:~/.../Labsetup$ tail -c 128 out1.bin > P
[11/26/23]seed@VM:~/.../Labsetup$ tail -c 128 out2.bin > Q
```

Now I can create two versions of the same program. Basically, when I concatenate the prefix, P or Q, and the suffix I can create two different programs with the same MD5 hash value. The following image shows the concatenation of the files to create two executable files (a1.out & a2.out) which differ but have the same hash value.

```
[11/26/23]seed@VM:~/.../Labsetup$ cat prefix P suffix > a1.out
[11/26/23]seed@VM:~/.../Labsetup$ cat prefix Q suffix > a2.out
[11/26/23]seed@VM:~/.../Labsetup$ chmod a+x a1.out a2.out
[11/26/23]seed@VM:~/.../Labsetup$ diff a1.out a2.out
Binary files a1.out and a2.out differ
[11/26/23]seed@VM:~/.../Labsetup$ md5sum a1.out a2.out
970395deb2a95e250b68ea016ca85a62  a1.out
970395deb2a95e250b68ea016ca85a62  a2.out
```

Now after running both programs you can see they both output different values:

```
[11/26/23]seed@VM:~/.../Labsetup$ ./a1.out
414141414141414141414141414141414141414141414141414141414141419026b7a64887c162
42f8fbcc7ef83884287377d7b2a105fb27c7669b8b1dae4adbedebb641c9f71ecde4fc9b97b3c0df
b1b31b4be2c92674587caa89d5522f579b8ffae2283235ea719ee1f35de63f545248db3702b28999
36cc99a1dde547fa6922bfe3c64e9d4d6a8dc4911ee320439bd0d5a393e79d8f759838d775414141
414141414141414141414141414141414141414141414141414141414141414141414141414141
[11/26/23]seed@VM:~/.../Labsetup$ ./a2.out
414141414141414141414141414141414141414141414141414141414141419026b7a64887c162
42f8fbcc7ef83884287b77d7b2a105fb27c7669b8b1dae4adbedebb641c9f71ecde4fc1b98b3c0df
b1b31b4be2c9267457caa89d5522f579b8ffae2283235ea719ee1f35de63f545a48db3702b289993
6cc99a1dde547fa6922bfe3c64e9d4d6a8d44911ee320439bd0d5a393e79d8ff59838d7754141414
141414141414141414141414141414141414141414141414141414141414141414141414141414
```

If you look closely at the right-end and second line of the output you can see the differences; look for c9 and c1 in the result from a1.out and a2.out respectively.

Conclusion

In conclusion, One-way hash functions are essential to cryptography and provide for a range of applications, such as blockchains, password authentication, etc. They have two properties: one-way and collision-resistant. I've learned a lot about the one-way hash functions and their properties. Learning about the length extension attack and the collision attack has given me the ability to spot and prevent such vulnerabilities. I now know what hashing algorithms are secure (SHA-2) and which ones are not (MD), and can use my new found knowledge to effectively implement secure hashing algorithms in programs.