MD5 Collision Lab

1 Overview

A secure one-way hash function needs to satisfy two properties: the one-way property and the collision-resistance property. The one-way property ensures that given a hash value h, it is computationally infeasible to find an input M, such that $hash(M) = h$. The collision-resistance property ensures that it is computationally infeasible to find two different inputs M_1 and M_2, such that $hash(M_1) == hash(M_2)$.

Several widely-used one-way hash functions have trouble maintaining the collision-resistance property. At the rump session of CRYPTO 2004, Xiaoyun Wang and co-authors demonstrated a collision attack against MD5. In February 2017, CWI Amsterdam and Google Research announced the SHAttered attack, which breaks the collision-resistance property of SHA-1. While many students do not have trouble understanding the importance of the one-way property, they cannot easily grasp why the collision-resistance property is necessary, and what impact these attacks can cause.

The learning objective of this lab is for students to really understand the impact of collision attacks, and see in first hand what damages can be caused if a widely-used one-way hash function's collision-resistance property is broken. To achieve this goal, students need to launch actual collision attacks against the MD5 hash function. Using the attacks, students should be able to create two different programs that share the same MD5 hash but have completely different behaviors. This lab covers a number of topics described in the following:

- One-way hash function
- The collision-resistance property
- Collision Attacks
- MD5

2 Lab Environment

This lab has been tested on the Ubuntu 16.04 VM that you should have installed on your computer during the first lab of this course.

3 Submission

Submit a PDF document with your answers to the tasks and questions in this lab. Include the task numbers and the questions. Place your answers immediately after the task number and question. Note that some answers require you to include a screen shot. Paste any code that you write or use next to the corresponding question. Some answers ask you to attach the source code to the Blackboard assignment. Use meaningful and easy to read names for your source code files.

4 Lab Tasks

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

In this task, we will generate two different files with the same MD5 hash values. The beginning parts of these two files need to be the same, i.e., they share the same prefix. We can achieve this using the `md5collgen` program, which allows us to provide a prefix file with any arbitrary content. The way how the program works is illustrated in the figure below. The following command generates two output files, `out1.bin` and `out2.bin`, for a given a prefix file `prefix.txt`:

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

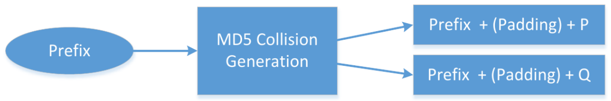

Figure 1: MD5 collision generation from a prefix

We can check whether the output files are distinct or not using the `diff` command. We can also use the `md5sum` command to check the MD5 hash of each output file. See the following commands

```
$ diff out1.bin out2.bin
$ md5sum out1.bin
$ md5sum out2.bin
```

Since `out1.bin` and `out2.bin` are binary, we cannot view them using a text-viewer program, such as `cat` or `more`; we need to use a binary editor to view (and edit) them. We have already installed a hex editor software called `bless` in our VM. Answer the following questions:

1. Suppose that you use an arbitrary prefix file that has a byte length that is not a multiple of 64 as your input to `md5collgen`. What will md5collgen do in terms of padding?
2. Suppose that you a prefix file that is 64 bytes long as input to `md5collgen` again. What happens in regards to padding in this case?
3. Create an arbitrary prefix file and generate two binaries with the same MD5 sum as explained above. The files generated by `md5collgen` are different. Which bytes are different and what are their values? You can use `bless` and/or `cmp` to find the answer.
4. The output files are different but they have the same MD5 sum. Identify the cryptographic hash property that is not met by the MD5 hash.
5. Identify an attack that can be launched due to the vulnerability that you identified above.

Deliverables. 1) Provide your answer to the three questions above. Include both the question and your answer in your submission document. 2) In the same file, provide your arbitrary prefix file and the output binaries in hex format from `md5collgen`. You may use screen shots for the binaries and prefix files, or simply copy and paste.

4.2 Task 2: Adding a suffix to two files that have the same hash value

In this task, we will try to understand some of the properties of the MD5 algorithm. These properties are important for us to conduct further tasks in this lab. MD5 is a quite complicated algorithm, but from very high level, it is not so complicated. As the figure below shows, MD5 divides the input data into blocks of 64 bytes, and then computes the hash iteratively on these blocks. The core of the MD5 algorithm is a compression function, which takes two inputs, a 64-byte data block and the output of the previous iteration. The compression function produces a 128-bit Intermediate Hash Value (IHV); this output is then fed into the next iteration. If the current iteration is the last one, the IHV will be the final hash value. The IHV input for the first iteration (IHV0) is a fixed value.

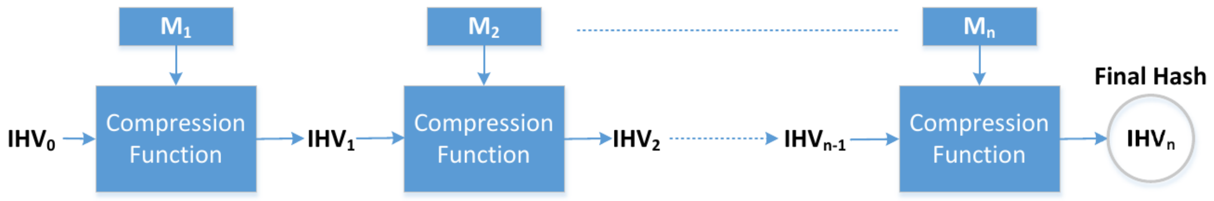

Figure 2: The MD5 Algorithm

Based on how MD5 works, we can derive the following property of the MD5 algorithm: Given two files 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.

That is, if inputs M and N have the same hash, adding the same suffix N to them will result in two outputs that have the same hash value. This property holds not only for the MD5 hash algorithm, but also for many other hash algorithms.

Your job in this task is to design an experiment that demonstrates this property holds for MD5. You can use the `cat` command to concatenate two files (binary or text files) into one. The following command concatenates the contents of `file2` to the contents of `file1`, and places the result in `file3`.

```
$ cat file1 file2 > file3
```

Deliverable. 1) A 1-2 sentences description of your experiment. 2) The files that you use and input files (i.e., M and N) and the file that you used concatenate (i.e., T). 3) The input files and their corresponding MD5 hash values that demonstrate the above.

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

In this task, you are given the following C program. Your 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.

```
#include <stdio.h>
unsigned char xyz[200] = {
/*The actual contents of this array are up to you*/
};

int main()
{
int i;
for (i=0; i<200; i++){
printf("%x", xyz[i]);
}
printf("\n");
}
```

You may choose to work at the source code level, i.e., generating two versions of the above C program, such that after compilation, their corresponding executable files have the same MD5 hash value. However, it may be easier to directly work on the binary level. You can put some random values in the `xyz` array and compile the above code to binary. Then you can use a hex editor tool to modify the content of the `xyz` array directly in the binary file. Finding where the contents of the array are stored in the binary is not easy. However, if we fill the array with some fixed values, we can easily find them in the binary. For example, the following code fills the array with `0x41`, which is the ASCII value for letter A. It will not be difficult to locate 200 A's in the binary.

```
unsigned char xyz[200] = {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,
  ... (omitted) ...
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  }
```

Guidelines From inside the array, we can find two locations, from where we can divide the executable file into three parts: a prefix, a 128-byte region, and a suffix. The length of the prefix needs to be multiple of 64 bytes. See the figure below for an illustration of how the file is divided.

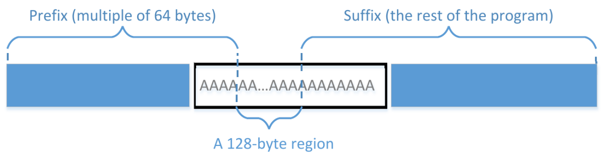
Prefix (multiple of 64 bytes)
Suffix (the rest of the program)
AAAAAA...AAAAAAAAAAA
A 128-byte region

Figure 3: Break the file into three parts

We can run `md5collgen` on the prefix to generate two outputs that have the same MD5 hash value. Let us use P and Q to represent the second part (each having 128 bytes) of these outputs (i.e., the part after the prefix). Therefore, we have the following: $MD5(prefix||P) = MD5(prefix||Q)$. Based on the property of MD5, we know that if we append the same suffix to the above two outputs, the output will also have the same hash value. Basically, the following is true for any suffix:

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

.

Therefore, we just need to use P and Q to replace 128 bytes of the array (between the two dividing points), and we will be able to create two binary programs that have the same hash value. Their outputs are different, because they each print out their own arrays, which have different contents.

Tools. You can use `bless` to view the binary executable file and find the location for the array. For dividing a binary file, there are some tools that we can use to divide a file from a particular location. The `head` and `tail` commands are such tools. You can look at their manuals to learn how to use them. We give three examples in the following:

```
$ head -c 3200 a.out > prefix
$ tail -c 100 a.out > suffix
$ tail -c +3300 a.out > suffix
```

The first command above saves the first 3200 bytes of `a.out` to `prefix`. The second command saves the last 100 bytes of `a.out` to `suffix`. The third command saves the data from the 3300th byte to the end of the file `a.out` to `suffix`. With these two commands, we can divide a binary file into pieces from any location. If we need to glue some pieces together, we can use the `cat` command. If you use `bless` to copy-and-paste a block of data from one binary file to another file, the menu item ”Edit -¿ Select Range” is quite handy, because you can select a block of data using a starting point and a range, instead of manually counting how many bytes are selected.

Note. `md5collgen` concatenates the prefix to the new blocks in the output files.

Deliverables. 1) The ending byte position of your prefix and the starting byte position of your suffix in the original C program; 2) A screen shot showing that the output of `md5sum` for both of your programs (i.e., the ones that are built with the two outputs from `md5collgen`) is the same. 3) The output of `cmp` for both of your programs showing that they are different. 4) The output of your programs showing that they still run and print the array to the screen.

4.4 Task 4: Making the Two Programs Behave Differently

In the previous task, we have successfully created two programs that have the same MD5 hash, but their behaviors are different. However, their differences are only in the data they print out; they still execute the same sequence of instructions. In this task, we would like to achieve something more significant and more meaningful.

Assume that you have created a software which does good things. You send the software to a trusted authority to get certified. The authority conducts a comprehensive testing of your software, and concludes that your software is indeed doing good things. The authority will present you with a certificate, stating that your program is good. To prevent you from changing your program after getting the certificate, the MD5hash value of your program is also included in the certificate; the certificate is signed by the authority, so you cannot change anything on the certificate or your program without rendering the signature invalid.

You would like to get your malicious software certified by the authority, but you have zero chance to achieve that goal if you simply send your malicious software to the authority. However, you have noticed that the authority uses MD5 to generate the hash value. You got an idea. You plan to prepare two different programs. One program will always execute benign instructions and do good things, while the other program will execute malicious instructions and cause damages. You find a way to get these two programs to share the same MD5 hash value.

You then send the benign version to the authority for certification. Since this version does good things, it will pass the certification, and you will get a certificate that contains the hash value of your benign program. Because your malicious program has the same hash value, this certificate is also valid for your malicious program. Therefore, you have successfully obtained a valid certificate for your malicious program. If other people trusted the certificate issued by the authority, they will download your malicious program.

The objective of this task is to launch the attack described above. Namely, you need to create two programs that share the same MD5 hash. However, one program will always execute benign instructions, while the other program will execute malicious instructions. In your work, what benign/malicious instructions are executed is not important; it is sufficient to demonstrate that the instructions executed by these two programs are different.

Guidelines. Creating two completely different programs that produce the same MD5 hash value is quite hard. The two hash-colliding programs produced by `md5collgen` need to share the same prefix; moreover, as we can see from the previous task, if we need to add some meaningful suffix to the outputs produced by `md5collgen`, the suffix added to both programs also needs to be the same. These are the limitations of the MD5 collision generation program that we use. Although there are other more complicated and more advanced tools that can lift some of the limitations, such as accepting two different prefixes, they demand much more computing power, so they are out of the scope for this lab. We need to find a way to generate two different programs within the limitations. There are many ways to achieve the above goal. We provide one approach as a reference, but students are encouraged to come up their own ideas (bonus points are available if you find a different approach). In our approach, we create two arrays X and Y. We compare the contents of these two arrays; if they are the same, the benign code is executed; otherwise, the malicious code is executed. See the following pseudo-code:

```
Array X; Array Y;

main(){
if(X's contents and Y's contents are the same)
run benign code;
else
run malicious code;
return;
}
```

We can initialize the arrays X and Y with some values that can help us find their locations in the executable binary file. Our job is to change the contents of these two arrays, so we can generate two different versions that have the same MD5 hash. In one version, the contents of X and Y are the same, so the benign code is executed; in the other version, the contents of X and Y are different, so the malicious code is executed. We can achieve this goal using a technique similar to the one used in Task 3

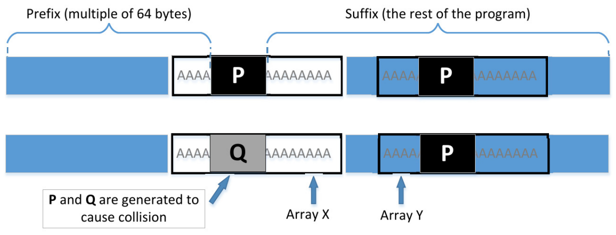
Prefix (multiple of 64 bytes)
Suffix (the rest of the program)
AAAA
P
AAAAAAAA
AAAA
P
AAAAAAA
AAAA
Q
AAAAAAAA
AAAA
P
AAAAAAA
P and Q are generated to cause collision
Array X
Array Y

Figure 4: An approach to generate two files with the same hash but different behaviors

From the figure above we know that these two binary files have the same MD5 hash value, as long asPandQare generated accordingly. In the first version, we make the contents of arrays X and Y the same, while in the second version, we make their contents different. Therefore, the only thing we need to change is the contents of these two arrays, and there is no need to change the logic of the programs.

Deliverables. 1) The C code that implements the pseudo-code. 2) The benign program. 3) The malicious program. 4) Screen shot of the output of `md5sum` for the binaries of the benign and malicious programs; 6)

Screen shot of the output of `cmp` to compare the contents of the malicious and benign binaries; 6) Screen shot of the output of the malicious and benign programs.