MD5 Collision Attack

1 Introduction

Cryptographic hash functions are required to be one-way and collision-resistant to be suitable for security applications. Newer families of cryptographic hash functions such as SHA-2 and SHA-3 currently fulfill these requirements. However, known attacks against older hash functions such as MD5 and SHA-1 render them unable to fulfill these requirements. In this lab, we will find MD5 hash collisions and demonstrate the security impact that collision attacks can have.

Readings. Coverage of cryptographic hash function can be found in the Week 8 lecture slides and in Chapter 11 of *Understanding Cryptography* by C. Paar and J. Pelzl, available at: `https://link-springer-com.proxy.queensu.ca/content/pdf/10.1007/978-3-642-04101-3.pdf`.

Lab Environment. This lab has been tested on our pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website. We will use the "Fast MD5 Collision Generation" program written by Marc Stevens, located at `/usr/bin/md5collgen` in the VM. You may also clone and compile the following repository on your own computer: `https://github.com/cr-marcstevens/hashclash`. After compiling the project, the program will be located in `bin/md5_fastcoll` in the repository directory.

Acknowledgment. This lab was developed with the help of Vishtasp Jokhi, a graduate student in the Department of Electrical Engineering and Computer Science at Syracuse University.

2 Submission

Please submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. Please answer the questions in detail, and explain any observations you make that are interesting or surprising. Please also explain any code that you write. Code submitted without any accompanying explanations will result in grade deductions.

3 Lab Tasks

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

In this task, we will generate two different files that have the same prefix (i.e., starting bytes) and that have the same MD5 hash values, as illustrated in Figure 1. Provide a prefix file `pre.dat` containing any arbitrary content as input to `md5collgen` and generate two output files `out1.bin` and `out2.bin` as follows.

```
$ md5collgen -p pre.dat -o out1.bin out2.bin
```

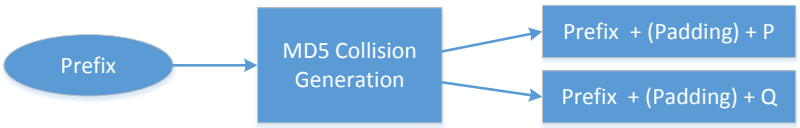

Figure 1: MD5 collision generation from a prefix

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 text-based tools such as `cat` or `more`. Instead, use a hex editor (e.g., `bless` has a GUI and is installed on the VM) to view the two output files and answer the following questions:

1. What happens if the length of your prefix file is not multiple of 64?
2. What happens when you provide the collision tool with a prefix file that is exactly 64 bytes?
3. Are the 128 bytes of new data generated by `md5collgen` completely different for each of the two output files? Identify the differing bytes.

3.2 Task 2: Understanding MD5

In this task, we will investigate some properties of the MD5 algorithm. As shown in Figure 2, MD5 divides the input data into 64-byte blocks 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 IHV, which stands for "Intermediate Hash Value"; 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 (IHV_0) is a fixed value.

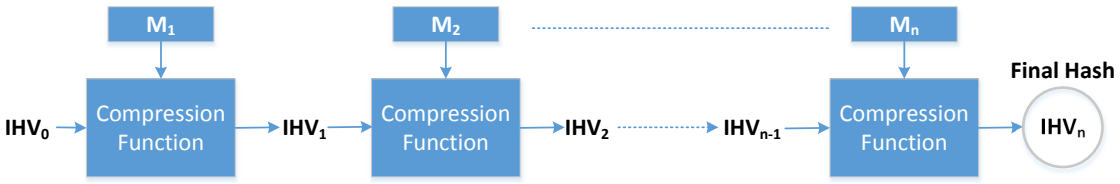

Figure 2: How the MD5 algorithm works

Observe the following: Given two inputs `M` and `N`, if `MD5(M) = MD5(N)`, i.e., if `M` and `N` have the same MD5 hash, then for any input `T`, `MD5(M || T) = MD5(N || T)`, where || represents concatenation. In other words, if inputs `M` and `N` have the same hash, adding the same suffix `T` to them will result in two outputs that have the same hash. This property holds for many other hash algorithms as well.

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

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

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

In this task, your job is to create two different versions of the C program below such that the contents of their `xyz` arrays are different, but the hash values of the resulting executable files 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");
}
```

One approach is to prepare two versions of the above source code that compile to two different executable files with the same MD5 hash. However, it can be easier to first compile the source code and then modify the contents of the `xyz` array in the compiled binary. To make it easier to locate the contents of the array in the compiled binary, we can modify the source code to fill the array with some known values. For example, the following code fills the array with `0x41`, which is the ASCII value for the letter `A`. We can then locate the 200 `A` characters in the compiled 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,
  ... (omitted) ...
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
}
```

Guidelines. Inside the array, we can identify two locations from where we can split 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 Figure 3 for an illustration of how the file will be split.

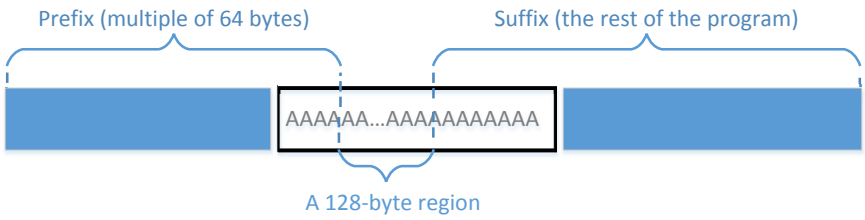

Figure 3: Break the executable file into three pieces.

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)
```

We know from the previous task that if we append the same suffix to the above two outputs, the resultant data 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 outcomes 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. To split the executable file, the `head` and `tail` commands will be useful (consult their manuals by typing `man head` and `man tail`). We provide three example usages below:

```
$ 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 `3300`th byte to the end of the file `a.out` to `suffix`. With these two commands, we can split a binary file into pieces from any location. Use the `cat` command if you need to concatenate any of the pieces together.

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.

3.4 Task 4: Making the Two Programs Behave Differently

In the previous task, we created two programs that have the same MD5 hash. However, the two programs differ only in what they print to the screen. In this task, we will create two programs with the same hash that differ more substantially in their functionality.

Consider an operating system that only allows the installation of applications that are cryptographically signed by the operating system vendor. Developers must submit their applications to undergo vetting and then receive a certificate signed by the vendor. The hash value of the application is embedded in the certificate, so neither the certificate nor the application can be modified without rendering the signature invalid.

If you wish to distribute a malicious application, you would have little chance of getting certified if you submit the malicious application for approval. However, you notice that the app store uses MD5 to generate hashes, and you decide to prepare two different applications that share the same MD5 hash value: One application will execute benign functionality, and the other will execute malicious functionality. You then submit the benign application for approval, and receive a certificate. Since the certificate contains the hash of your benign application, and your malicious application has the same hash, the certificate is also valid for your malicious application. This allows you to distribute your malicious application to unsuspecting users.

Your task is to launch the attack described above. Namely, you need to create two programs that share the same MD5 hash. One program will always execute benign code, while the other program will execute malicious code. 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 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 are using. Although there are more advanced tools with fewer limitations, they demand much more computing power, so we choose to explore approaches that work within this limitation. We provide one reference approach below, but you may also consider coming up with your own 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 pseudocode:

```
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. We can then change the contents of these two arrays to end up with two different programs 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. Figure 4 illustrates what the two versions of the program look like.

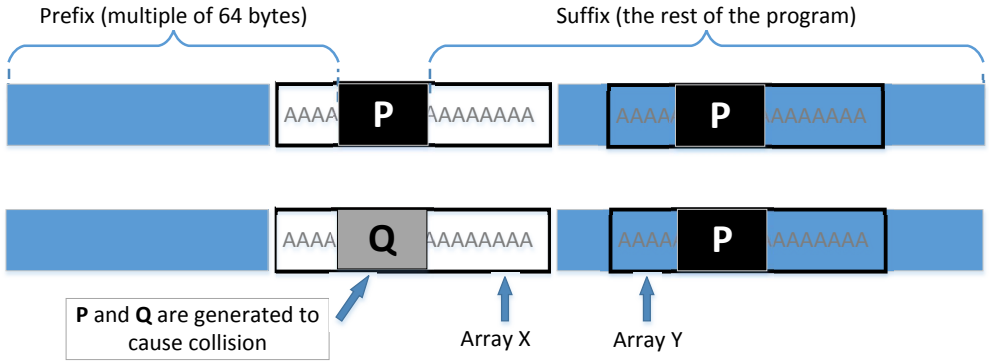

Figure 4: An approach to generate two hash-colliding programs with different behaviours.

From Figure 4, we know that these two binary files have the same MD5 hash value, as long as `P` and `Q` are 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.