**Lab 6 Collision Attack Lab**

**1. Goal and Deliverable**

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.

Please attach all required screen shots and answer all of the questions, then submit the lab report to D2L.

**2. 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 M1 and M2, such as hash(M1) = hash(M2).

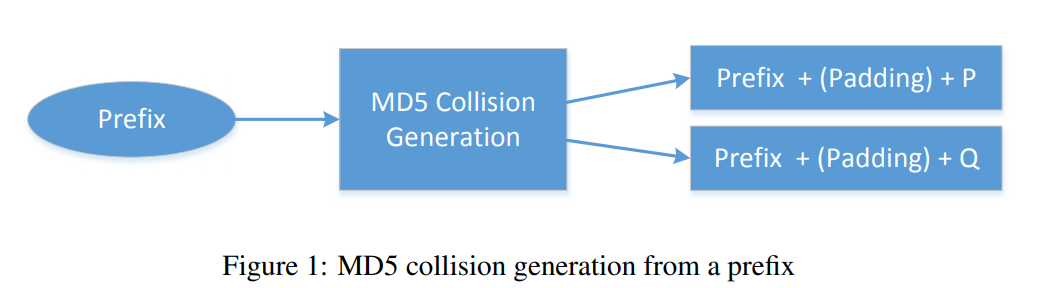
Several widely-used one-way hash functions have trouble maintaining the collision-resistance property. At the rump session of CRYPTO2004, Xiaoyun Wang and co-authors demonstrated a collision attack against MD5 [1]. In Febrary 2017, CWI Amsterdam and Google Research amounted the SHAttered attack, which breaks the collision-resistance property, they cannot easily grasp why the collision-resistance property is necessary, and what impact these attacks can cause.

**3. Lab Tasks**

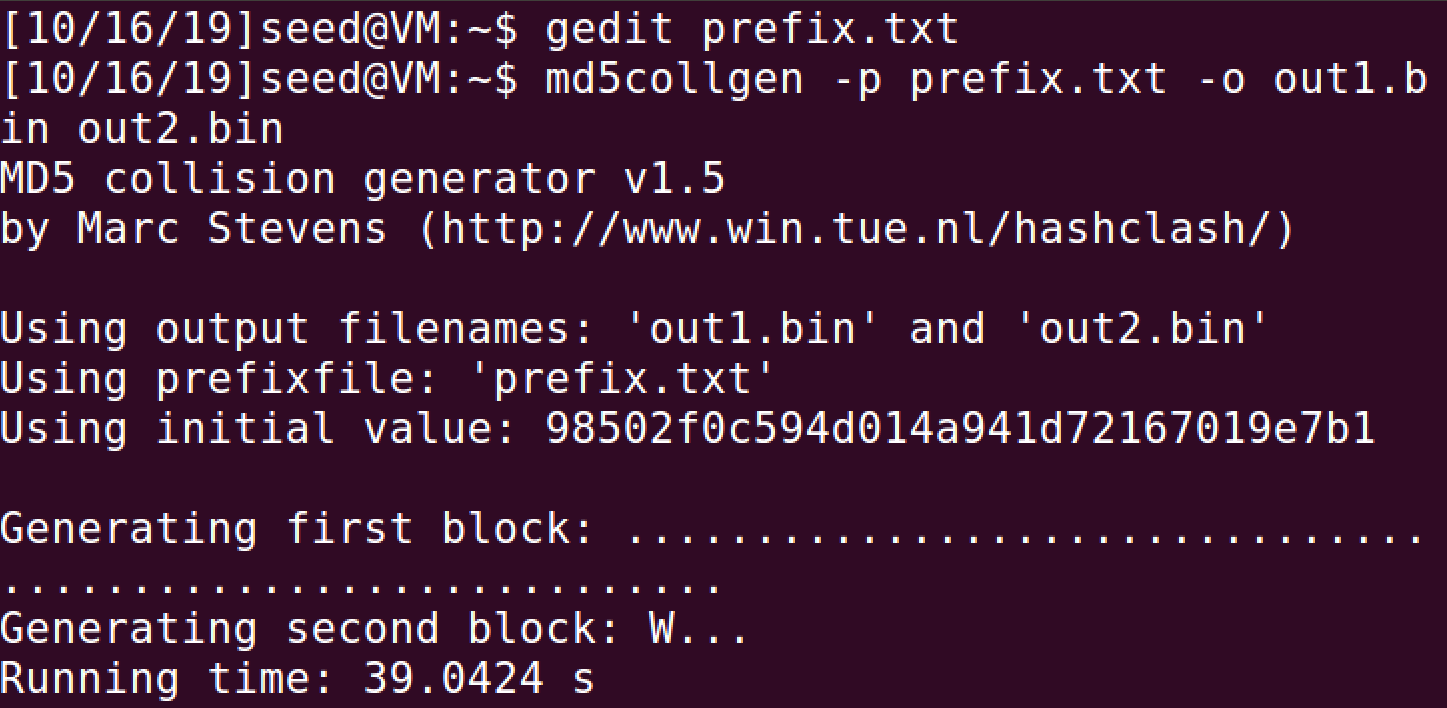
**3.1 Task 1: Generating Two Different Files with the Same MD5 Hash**

In this task, we will generate two different files 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 Figure 1. The following command generates two output files, out1.bin and out2.bin, for a given prefix file prefix.txt:





Step 1: creating a prefix.txt file with any arbitrary content (I used CSC 302 Computer Security as the input), then run the above command. You will get similar output as shown in the following figure.



Step 2: check the hash code

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.



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. Please use such an editor to view these two output files with command bless out1.bin and bless out2.bin.

Please attach a screen shot for the hash value of out1.bin and out2.bin.

A screen shot of a computer

Description automatically generated

Question1: If the length of your prefix file is not multiple of 64 bytes, what is going to happen?

**It will be padded with zero’s.**

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

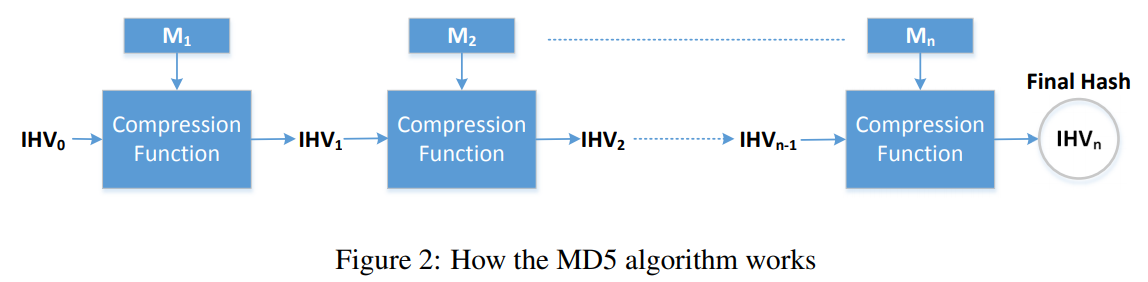
**There is no padding, and the runtime of “md5collgen” is significantly faster.**

Hint1: You can use command du –sh out1.bin to check the file size.

Hint2: how to create a file with certain size in Linux: <https://www.ostechnix.com/create-files-certain-size-linux/>

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

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 Figure 2 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 outcome 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 (IHV0) is a fixed 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.

That is, 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 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 to demonstrate that this property holds for MD5. Please show screen shots to verify the results.

A screenshot of a computer

Description automatically generated

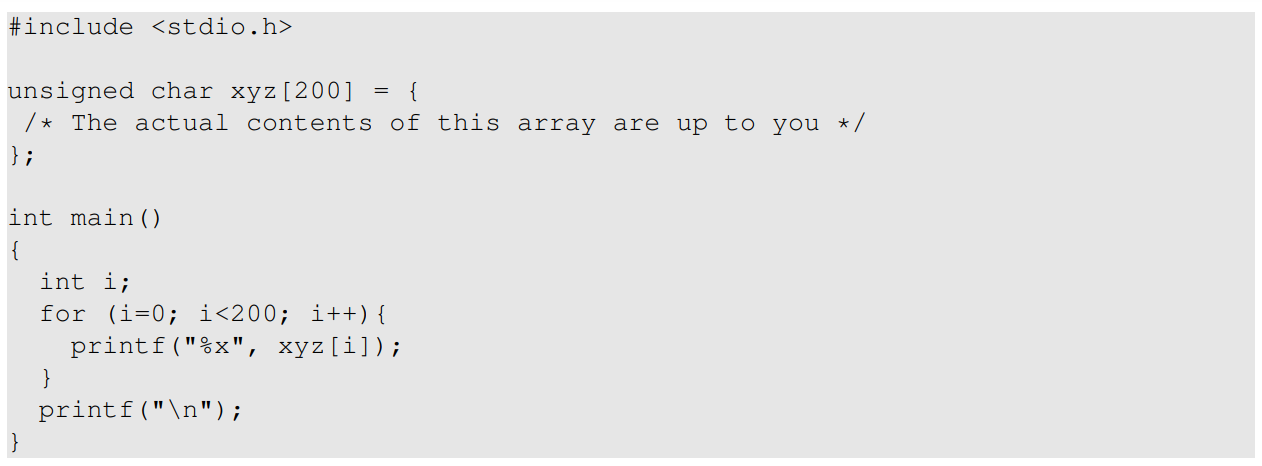
Hint: You can use the cat command to concatenate two files (binary or text files) into one. The following command concatenate the contents of file2 to the contents of file1, and places the result in file3.



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

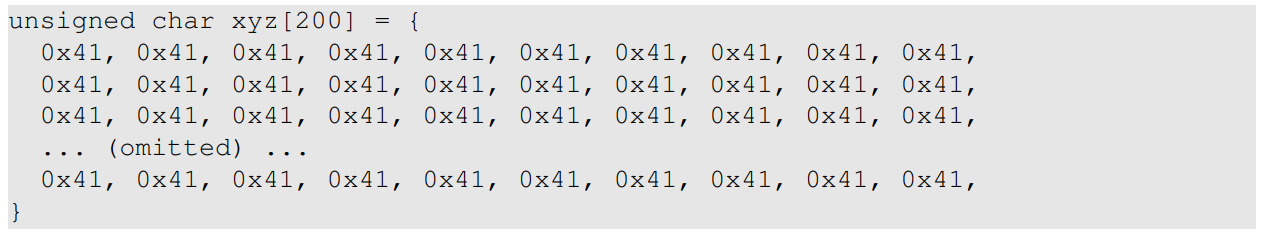
In this task, you are given the following C program.

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.

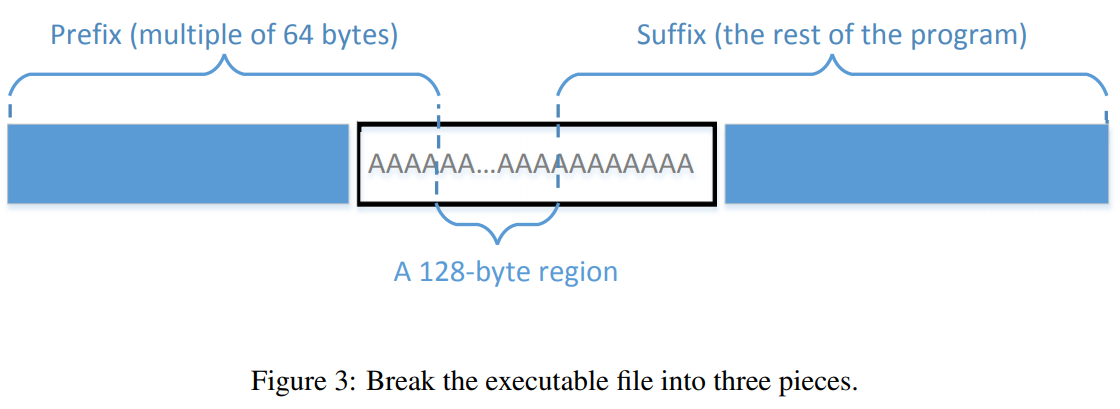


You may choose to work at the source code level, ie.e, generating two versions of the above C program, such that after compilation, their corresponding executable files have the same MD5 hash value. However, it may be easier to directly work on the binary level. You can put some random values in the xyz array, 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.



**Guidelines.** From inside the array, we can find two locations, from where we can divid 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 is divided.



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:

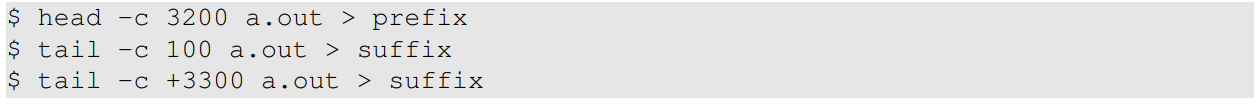


Based on the property of MD5, we know that if we append the same suffix to the above two outputs, the resultant data will also have the same hash value. Basically, the following is true for any suffix:



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. For dividing the 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 useful tools. You can look at their manuals to learn how to use them. We give three examples in the following:



The first command above saves the first 3288 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 three commands, we can divide a binary file into pieces from any location. If you need to glue some pieces together, we can use the cat command.

Please attach screenshots on your two programs, as well as the screenshot that show both of them have the same hash value.

**3.4 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 MD5 hash 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 go 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.

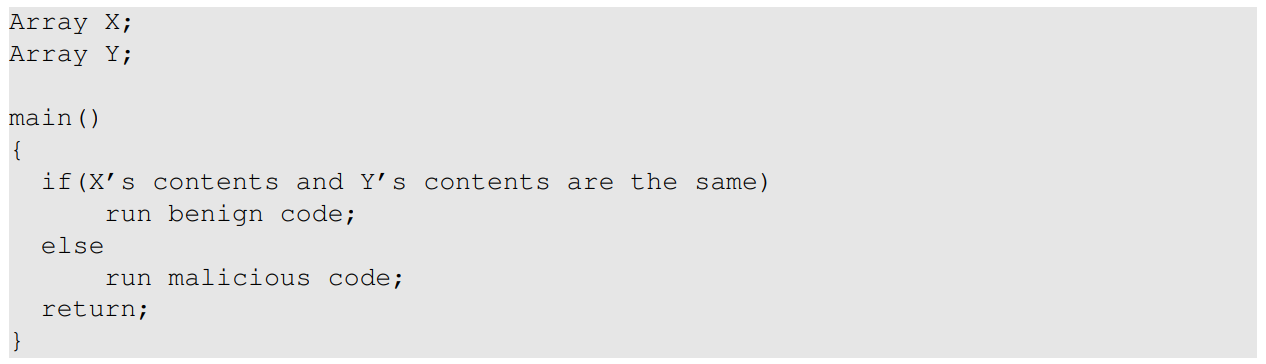
The objective of this task is to launch the attack described above.

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 program are different.

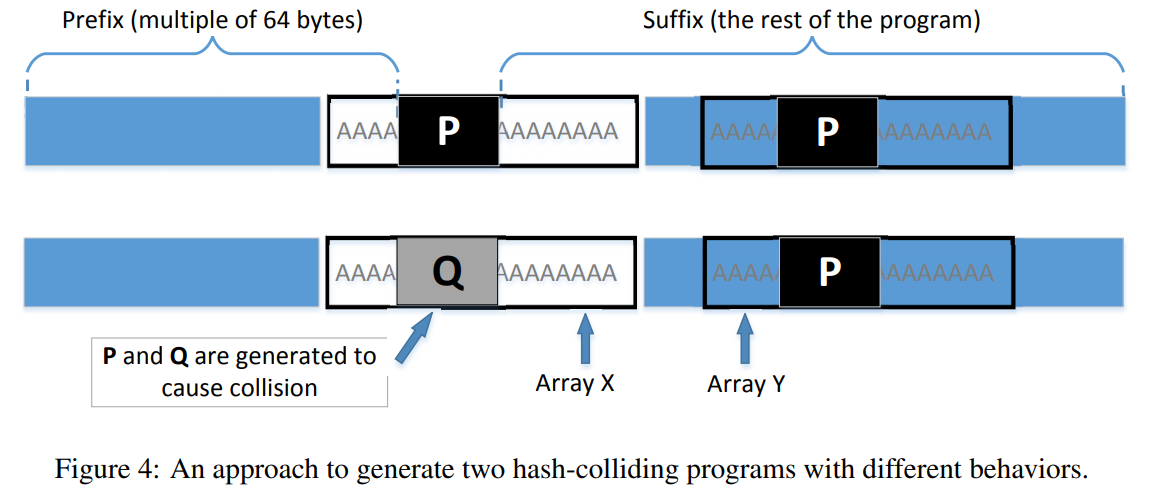
Please attach screenshots for both of your programs and the hash code to demonstrate even if they have different operations, the MD5 hash is the same.

**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 [2], 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 you are encouraged to come up different ideas. In this 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:



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. Figure 4 illustrates what the two version of the program look like.



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.