Hash Attacks[1]

Due Friday, April 15th, 2022 @ 11:59pm

(**8%** of the total course grade)

1. Objective

A secure one-way hash function needs to satisfy two important 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 [1]. In February 2017, CWI Amsterdam and Google Research announced the *SHAttered* attack, which breaks the collision-resistance property of SHA-1 [3]. 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 purpose 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 will be graded. It has to be completed INDIVIUALLY, but you may want to discuss the lab content with your fellow students. After you have finished the lab exercises, please submit your lab report before the due date. You must demonstrate your lab in your report through a step by step MOP (Method of procedure).

Also, a sample lab report written in an Attack and Defence course by my former student at the University of Ontario Institute of Technology is provided for your reference.

2. Environment Setup

If you haven't set up your virtual Cyber Security Lab environment using VirtualBox on your laptop, please install and setup VirtualBox and Ubuntu 16.04 virtual machines by following instructions provided in **Lab 1**.

Please note, for Lab 4, you only need to use one Ubuntu 16.04 VM.

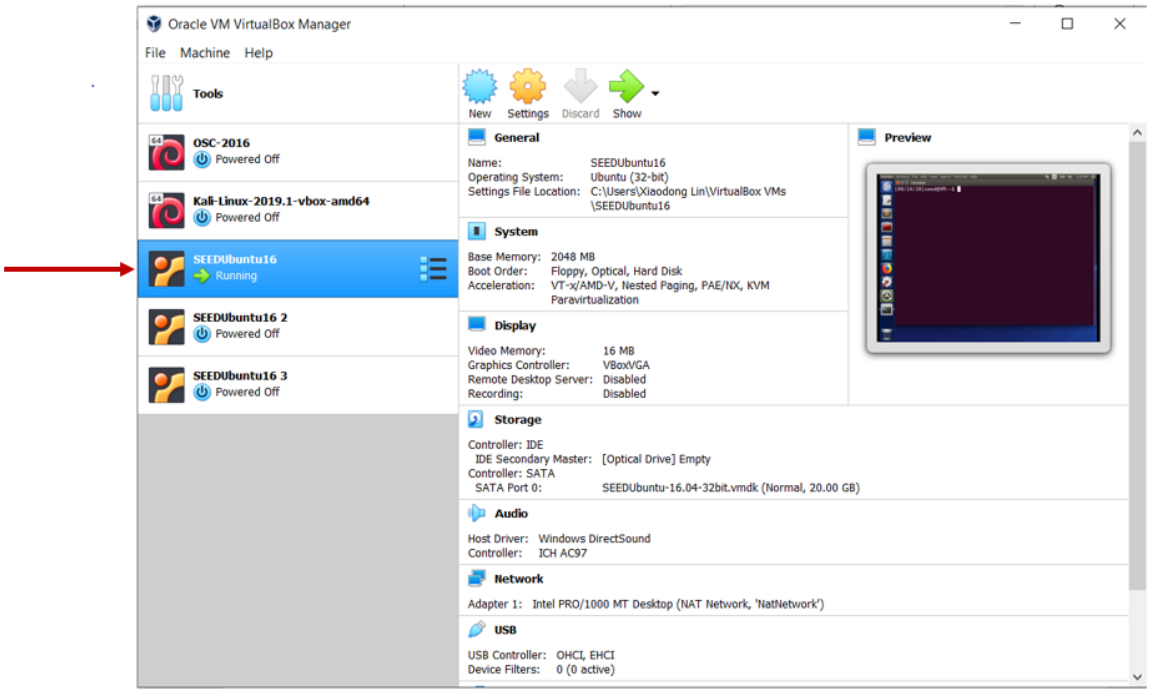

Figure 1. Required Lab Environment Settings

3. Lab Tasks

3.1 Task 1: Calculate MD5, SHA-1 and SHA-256 hash values of the file t3.c

In this task, we will use the utilities md5sum, sha1sum, and sha256sum to generate the MD5, SHA1 and SHA-256 hash values of the file t3.c.

- **Q1:** What is the MD5 hash value of the file t3.c? (0.5 mark)
- **Q2:** What is the SHA-1 hash value of the file t3.c? (0.5 mark)
- **Q3:** What is the SHA-256 hash value of the file t3.c? (0.5 mark)

3.2 Task 2: 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 2. 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
```

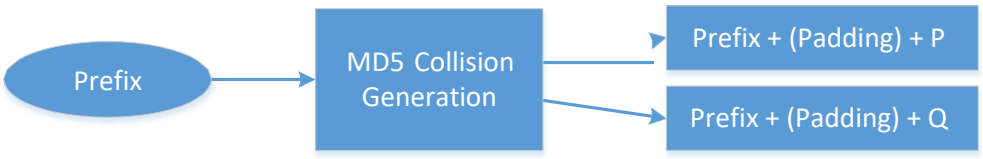

Figure 2: MD5 collision generation from a prefix

We can check whether the output files are distinct or not using the diffcommand. We can also use the md5sumcommand 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.binand out2.binare 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, and describe your observations. In addition, you should answer the following questions:

- **Q4.** If the length of your prefix file is not multiple of 64, what is going to happen? (1 mark)
- **Q5.** Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens. (1 mark)
- **Q6.** Are the data (128 bytes) generated by md5collgen completely different for the two output files? Please identify all the bytes that are different. (1.5 marks)

Please note: As for Tasks 3 and 4, you will need to describe your experiment procedures and results in your report with all support materials (important screenshots and source codes where applicable (e.g., the shell script you have developed).

3.3 Task 3: Generating Two Executable Files with the Same MD5 Hash (2 marks)

In this task, you are given the following C program named **t3.c**. Your job is to create two different versions of this program, such that the contents of their **a** arrays are different, but the hash values of the executables are the same.

```
#include <stdio.h>

unsigned char a[200] =
{ 'A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A'
,'A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A',
'A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','
A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A
','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A'
,'A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A',
'A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','
A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A
','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A'
,'A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A','A',
'A','A','A','A','A','A','A','A','A'};

int main()
{
       int i;
       for(i = 0; i < 200; i++)
       {
              printf("%x", a[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 **a** array, compile the above code to binary. Then you can use a hex editor tool to modify the content of the **a** 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 code named **t3.c** provided to you fills the array with the letter **A**. It will not be difficult to locate 200 **A**'s (or **0x41**) in the binary.

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 Figure 3 for an illustration of how the file is divided.

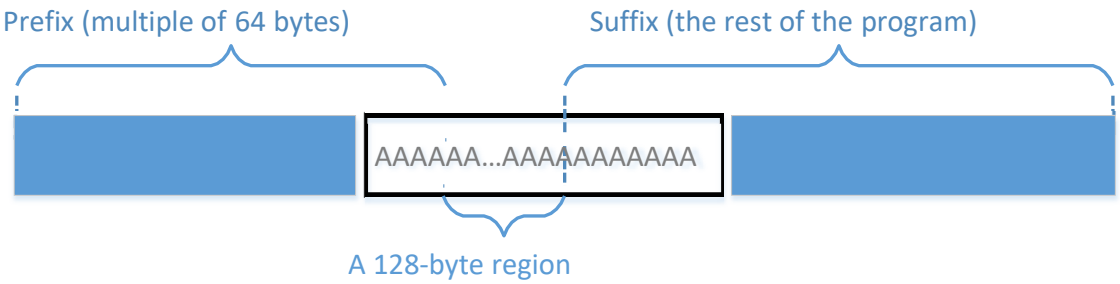

Figure 3: Break the executable file into three pieces.

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)
```

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:

```
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. 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 useful 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 `3300`th byte to the end of the file `a.out` to `suffix`. With these 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.

3.4 Task 4: Making the Two Programs Behave Differently (3 marks)

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 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**.

You are provided with a c program t4.c. It creates two arrays `a` and `b`. 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 code below:

Since the arrays `a` and `b is` initialized with 'A' 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 a and b are the same, so the benign code is executed; in the other version, the contents of a and b are different, so the malicious code is executed. We can achieve this goal using a technique similar to the one used in Task 3.

```
#include <stdio.h>

unsigned char a[400] =
{ "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA"};
unsigned char b[400] =
{ "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA"};

int main()
{
	int i;
	int isSame=1;
	for(i = 0; i < 400; i++)
	{
		if(a[i]!=b[i])
			isSame=0;
	}
	if(isSame)
		printf("This would run the safe code and display the intended
behaviour");
	else
		printf("This is where malicious code would be run");
	printf("\n");
}
```

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

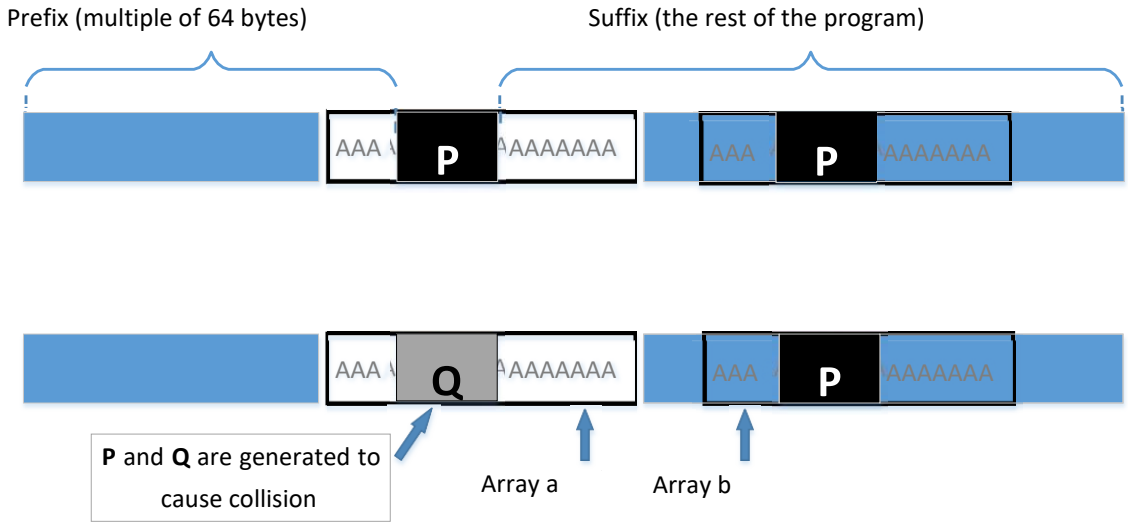

Figure 4: An approach to generate two hash-colliding programs with different behaviors.

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 a and b 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.

4. Final Lab Report

After you have done with your lab experiments, you need to submit a laboratory report to me by describing the experiment procedures and results with all support materials including your answers to the questions in the lab , important screenshots and source codes where applicable. The report must be named lab4_XXX_YYY.pdf, where XXX is your student ID and YYY is your surname in Pinyin. This naming convention facilitates the tasks of marking for the instructor and course TA. It also helps you in organizing your course work. Failure to follow the requirements will result in mark reduction.

It is very important to include screenshots which demonstrate that you have successfully completed the tasks required in the lab.

What is a good lab report to me? A good report is that if I use your report, it should be easy for me to set up and conduct the experiment and obtain the same results that you did. In other words, your experiment should be repeatable to me.

Acknowledgements

This lab has incorporated the materials developed by Dr. Wenliang Du (Syracuse) and Dr. J. Alex Halderman (University of Michigan). The copyright of these materials belongs to them.

Reference:

[1] John Black, Martin Cochran, and Trevor Highland. A study of the md5 attacks: Insights and improve- ments. In Proceedings of the 13th International Conference on Fast Software Encryption, FSE'06, pages 262–277, Berlin, Heidelberg, 2006. Springer-Verlag.
[2] Marc Stevens. On collisions for md5. Master's thesis, Eindhoven University of Technology, 6 2007.
[3] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. CWI Amsterdam and Google Research, https://shattered.io/, 2017.
[4] Mike Rosulek. The Joy of Cryptography
https://web.engr.oregonstate.edu/~rosulekm/crypto/