

C0D3 CR4CK3D: Means and Methods to Compromise Common Hash Algorithms

Kevin C. Redmon, Graduate Student in Information Security, East Carolina University

Abstract — There are currently several different hashing algorithms in use today. These include LM, MD5, SHA-1, and SHA-2, among many others. We use these algorithms for many different purposes – data fingerprinting, digital signatures, and message authentication codes. In these applications, there is one common element – the hashing algorithm takes a piece of data that is likely bigger in size and reduces it to a shorter, “unique” identifier, called a hash. If the data changes in a slight way, the resulting hash can change quite drastically, some times impacting ~50% of the bits in the resulting hash [1]. Furthermore, to create the same hash from two different data elements becomes increasingly more difficult as the length of the hash increases. These algorithms and the resulting hash, were designed to provide security and confidence to the user that the data is authentic.

What happens when two different data elements create the same hash? The entire security and usefulness of the hashing algorithm falls under scrutiny. Simply proving that a particular hash algorithm can be compromised in our lifetime can also create enough alarm to send cryptographers back to the drawing board. Many cryptanalysts see this as a challenge - to “break the unbreakable” - and dedicate their lives to break these algorithms. In this paper, I will discuss the means and methods that cryptanalysts use to compromise several hash algorithms. I will also discuss ways to decrease the opportunity for a compromise of a hash or its source data.

Index Terms—Information Security, Cryptography, Hash, Cracking

I. INTRODUCTION

HASHING has long been used as a means to verify data elements. Parity bits were originally used to confirm that a data transmission was received correctly and helped to detect any single-bit errors. However, parity didn’t add any value if multiple bits in the data had errors. As a result, a second trend then came about called CRC – Cyclic Redundancy Checks. These CRCs, based on polynomials, were used to detect errors in the data elements via a hash. Although this approach is more robust than parity bits, weaknesses in this algorithm also came to pass. A user could modify a file and easily sculpt the file’s contents to create the same CRC as the original file [5]. As such, we needed

another way to verify our data files.

In comes hashing as we know it today. The modern hashing algorithms are designed to help verify the integrity of a data element. Yet, as our needs to verify file contents and data transmissions evolve, we continually realize that better solutions and algorithms are needed. Better hashing algorithms are constantly in the works. However, as the past would indicate, cryptographers and hackers alike continue to find issues with each successive algorithm that is released. As one algorithm is broken, another is developed – so the cycle continues. This paper will discuss the means and methods used to compromise these hashing functions and ways to protect yourself, and your data.

II. HASH ALGORITHMS

There are approximately fourteen major hash algorithms in common use today[2]. Since some of these algorithms have fallen to disuse, I chose to address those that are most often in the spotlight – LM (LanManager), MD5, SHA-0/SHA-1, and SHA-2. (SHA-0 and SHA-1 are closely familiar; they are often approached as a single algorithm.). I will discuss the cryptographic functions of each of these hashing algorithms and the key contributor to their security.

LM, also known as LanManager, is a hashing algorithm developed by Microsoft in the early 1990’s. It was first used in their Windows 3.11 product and is not very secure [3]. This particular hashing function is used to “protect” your password on most Windows operating systems. Despite the existence of more secure hashing algorithms, including NTLMv2 and Kerberos, Microsoft continues to support this for legacy reasons. This algorithm is as follows [4]:

1. Convert the user’s password to uppercase.
2. Either null-pad the password or truncate to 14 bytes/characters.
3. Split the resulting password into 7-byte halves.
4. Create two DES keys, using the two 7-byte halves as inputs.
5. Each of these keys are used to DES-encrypt the constant “KGS!@#%”, resulting in two 8-byte ciphertext values.
6. Concatenate these two ciphertext values to form a 16-byte value – this is the LM Hash.

The weaknesses in this algorithm are many. First and foremost, the hash is case-insensitive. Secondly, the character set is limited to 142 characters. Finally, and the issue that is

Manuscript received July 17th, 2006.

Kevin Redmon is a Graduate Student studying Information Security at East Carolina University, East Fifth Street, Greenville, NC 27858 USA (e-mail: kcr1116@ecu.edu).

most exploited is the hash is broken down into two 7-byte chunks. The best security for this hashing algorithm will be to keep the resulting hash out of the hands of unauthorized people.

The second algorithm that I'd like to discuss is MD5 (Message Digest algorithm 5). This hashing function was created by Ronald Rivest, an MIT professor, in 1991 as a successor to MD4 [1,6]. The algorithm starts by padding the incoming data to result in a file length that is divisible by 512 bits. The algorithm then takes the data in 512-bit blocks. Each of these blocks is divided into sixteen 32-bit sub-blocks. Each of these sub-blocks is then used to affect the 128-bit state variable (divided into four 32-bit variables - a, b, c, d) that is used in the algorithm. In turn, the variables are processed through four rounds of four different nonlinear functions. Completing the fourth round, the final state (a', b', c', d') is added back to the original inputs from this round of the algorithm. After all of the data is processed, this 128-bit state variable remains.

This MD5 algorithm is significantly more secure than LM as mentioned above due to the nonlinear functions and a faster "avalanche" effect, yet is still vulnerable to compromise [1]. The key vulnerability for this algorithm lies in the length of its final hash. We'll discuss this more as we revisit the Types of Attacks in Section IV.

The SHA-0/SHA-1 algorithms are very similar in operation as MD5 [1]. The incoming data is padded to create a final size that is divisible by 512-bits. This incoming data is once again divided into sixteen 32-bit sub-blocks. In this algorithm, there is a 160-bit state variable (divided into five 32-bit variables - a, b, c, d, e). The algorithm consists of four rounds of 20 operations each, using the state variable and the incoming data as inputs. After all of the incoming data is processed, the 160-bit state variable remains.

The SHA-0/SHA-1's security comes from the use of the resulting 160-bit hash (vs. MD5's 128-bit hash), the addition of an expand transformation, and the use of the previous steps output into the current step. This last feature provides for a faster "avalanche" effect [1]. Some SHA-1 vulnerabilities have been found which will be discussed later in this paper.

SHA-2 variants include SHA-224, SHA-256, SHA-384, and SHA-512. The incoming data is once-again divisible by 512-bits. However, the data is either divided into 32-bit words (SHA-224/256) or 64-bit words (SHA-384/512). Also, the number of rounds, shift amounts, and additive constants vary slightly between each variety. The resulting output from SHA-2 variants is indicated by their suffix (SHA-256 has a 256-bit digest as its output) [7].

With SHA-2, its security is not yet well established as the variants have not fallen under the scrutiny of the cryptographic community [7]. However, assuming no tragic flaws in the algorithm, the sheer length of the resulting digests should make this algorithm a formidable opponent to any cryptanalyst.

MD5, SHA-0/SHA-1, and SHA-2 are of the Merkle-Damgård construction. These types of hash functions can accept an incoming data element of arbitrary-length and the resulting output will always be of a prescribed length [2].

With each of these algorithms, I will address some of their uses, types of attacks, and ways to protect your data from compromise while using the algorithm in question.

III. USES OF HASHING

Hashes are used for many different reasons. As I mentioned in the introduction above, hashing, in its simplest form, was used to detect errors in a file's transmission. Now, a hashing solution must do more than that - it needs to be more secure. Hash algorithms today have to offer a minimum level of security and assist in verifying the integrity of the data element.

One use of a hash function is for a digital signature. As the primary reason for the design of hash functions, the data to be signed is first ran through a hashing algorithm. This hashing algorithm reduces the data down to a manageable signature element. This signature element is then passed to a signature authority (ie RSA) to be digitally signed [8].

A second use of hashing is for Message Authentication Codes (MAC). MACs are one-way hash functions with the addition of a secret key [1]. The only way to verify a MAC value is to have the secret key. In some MAC algorithms, the hashing and use of the key happen multiple times, adding an additional layer of protection [8].

A third use of hashing, and possibly the most uncommon, is as a Pseudo-random function. By running a data element through a hashing algorithm, random-seeming bits are created. These can be used for generating cipher keying material after a Diffie-Hellman exchange [8].

A fourth, and probably the most common use of hashing, is data fingerprinting. By taking the data element and running it through a hashing algorithm, a "unique" identifier is created. This identifier can often be used to verify the integrity of the file. If a file is modified, even slightly, the resulting hash of the file will likely change quite drastically. A single bit change in a hashing algorithm input can sometimes impact 50% of the resulting hash.

IV. TYPES OF ATTACKS

Compromising the resulting hash of a data element is often the focus of many types of attacks on hashing algorithms. Although there are other compromises and attacks that can be done. The end result of most attacks is to gain access to a particular file or password. However, for a cryptanalyst, the end goal is to dissect and break the underlying algorithm.

We'll begin by classifying some of the attacks that are common with hashing [8]. First, a collision attack. A collision attack is where two files have the same resulting hash for a given algorithm. Given a particular file, a second file is designed so as to create the same hash. This is often the initial step in discounting the security of a hashing algorithm. A second attack is a preimage attack. This attack is often used to attack password hashes. The goal here is to find another input to the hash algorithm that will create the same hash as the original password. The third and final attack is called a second preimage attack. The goal here is to find two files that will create the same hash [8].

Executing a collision attack can have a varying degree of difficulty, depending on the original algorithm. Many cryptanalysts spend years trying to dissect the algorithms inner functions, searching for patterns or unseen vulnerabilities. One of the main tools cryptanalysts use are differential attacks, where the cryptanalyst investigates how a change in a single bit communicates through the rest of the hashing execution. In this case, there are two varieties of differential attacks – exclusive-or and modular integer subtraction [13]. The latter has become the method of choice for cracking modern encryption algorithms. Using this approach, Xiaoyun Wang et al. were able to expose a key vulnerability with the simple step operation function in SHA-0 and SHA-1 [14]. Following the cryptanalytic attacks on SHA-0/SHA-1, in August of 2004, the NIST announced that they will migrate to SHA-2 algorithms by 2010 [12].

For those collision attacks that are less scientific in nature, we refer to those as “brute-force” attacks. This attack method simply tests every possible outcome until the desired hash is found. The success opportunity can be roughly estimated by the Birthday Paradox, hence it is sometimes called a “birthday attack” [10]. This concept states that the probability for a collision (either in the birth date of a person, or extrapolated to estimate a hashing algorithm collision) can be estimated by the following formula:

$$p(n) = (N)(N-1)(N-2)...(N-n+1)/(N)^n$$

Where N = number of opportunities, n = opportunities taken, and p(n) = the probability of success [11]. If you want to determine the tipping point of this formula, where the probability of success is greater than 50%, you can estimate this using the following formula:

$$n(0.5) = 1.1774 \sqrt{N}$$

In this formula, n(0.5) is the number of attempts that you must make in order to have a greater than 50% chance at success. N is once again, the number of opportunities. In the calculation of collisions, either formula can be used to determine the average number of attempts until success. For instance, if a 64 bit hash is used, the number of different outputs is 2^{64} . Plugging this number into the formulas above, you will see that it will take nearly 5.1×10^9 to generate a collision.

A pseudo-collision can also indicate a possible vulnerability in the underlying algorithm. A pseudo-collision is when the original algorithm is modified from its specified state in order to create a collision situation. This can be accomplished by modifying the initial constants (Initialization Vectors) outside of those specified in the RFCs or reducing the number of rounds in the compression function of the algorithm [9]. This pseudo-collision attack is often the precursor to other attacks. World-renowned cryptanalyst, Bruce Schneier, was once quoted as saying “Attacks always get better; they never get worse.” [15]

Preimage attacks have been popular for over twelve years. The most prevalent use of preimage attacks were on UNIX-based systems. By gaining access to the password file, a user

could run a series of attacks on the password file to determine the password of one or more users. These password file contained user IDs and a hash of the password. The types of preimage attacks that are possible include brute-force attacks, dictionary attacks, and rainbow table attacks.

Brute force attacks, as above, are when every known possibility is tried. In this case, the hash of every known permutation is created and compared to that in the password file. Depending on the character set supported by the operating system in question and the maximum length of the password, the maximum number of attempts would equate to:

$$P = \sum C^L$$

Where L = 1 to Length of password and where C = the number of characters in the character set [16].

A dictionary attack is slightly more calculated, yet can also be time consuming. This attack will use a number of dictionary files and create hashes of each word and various permutations and combinations of the words, depending on the particular attack implementation. Depending on the password choice of the system user, this attack can either be very easy or very difficult. Choosing a word directly from the dictionary can result in an easy dictionary attack.

The third preimage attack is a Rainbow table attack. A rainbow table is a pre-computed group of tables that contain pre-hashed passwords (or other data elements). These files are then stored on your hard-drive until the time comes to use them. The tables are constructed in rainbow chains. Each chain will take a particular string and hash it. The resulting string is again hashed, creating another hash value. This is repeated t times. This creates what is known as a rainbow chain. The last value in each resulting chain is called L(t). The content of each rainbow table will be the first and last value of each chain (internal values on each chain can be recomputed as needed – yet we throw these values away to conserve disk space). The hash value to be cracked is then run through a similar process – hashing it over and over t times. Each of the values created by this hashing process, call it C(t), will be compared to the pre-computed values L(t) in the rainbow table. If a value (or multiple values) are found where C(t) = L(t), we can then recycle through those chains (remember, we know the beginning value of each chain) to find the value that hashes to our password. Significant research has been done by Philippe Oechslin and others, finding that a password can be found nearly 7.5 times faster using 12 times fewer calculations by using Rainbow tables versus computing the hashes in real-time [18]. This results in cracking most Windows passwords in an average of 66.3s (although, much faster times are possible with correct rainbow table selection).

A final type of attack is a second pre-image attack. This attack consists of creating two separate files that will result in the same hash with the intent of swapping one out of the digitally signed articles for the malicious version. This attack is minimally useful and depends heavily on the strength of the hashing algorithm.

V. PROTECTION

Now that we know how the level of security available in the various cryptographic hashes and how to attack them, we will now briefly review some methods to protect ourselves and our data.

The first way to protect yourself is to choose the right hashing algorithm. In one of the greater vulnerabilities mentioned in this paper, LM hashing in Windows, you find that LM passwords are easily decrypted. Due to the inherent weakness in this algorithm, attributed mostly to the length of the two 7-byte halves, it offers virtually no protection. Within Windows, you can either configure the operating system to disallow the use of this algorithm (despite its need for legacy applications) or select a password that is longer than 15 characters long. By choosing a longer password, whatever LM hash that is created will not match the real hash of your password and will, therefore, not authenticate the user [18].

A second way to protect yourself is to use “key strengthening”. Key strengthening a technique used to make a weak key stronger. This could be a password or a PIN number. During key strengthening, a “weak” key is ran through an algorithm that takes a known amount of time – preferably an algorithm that results in a key of a sufficient size (>128 bits). One example would be the SHA-1 algorithm. The SHA-1 algorithm can run 65000 iterations or more in one second on a modern PC. So, the weak key can be run through this algorithm, creating a much stronger key. Using the example above, this would effectively add about 16 bits of additional security to your password or original key. If we assume Moore’s law holds true (the computer speed doubles every 1.5 years), every 1.5 years an additional bit of key strength is possible to crack. This would theoretically mean an additional 24 years of security [19].

A third method to enhance the security of a cryptographic hash is “salt”. A salt acts as an initialization vector into the chosen hashing algorithm. Each bit of salt used doubles the amount of storage and computation required. The original Unix systems used a 12-bit salt, whereas modern implementations use more [20].

The final method in protecting yourself is to be smart when downloading files. As many of the vulnerabilities with hashing can be exploited with malicious intent, you should only download files from trusted sources. This will help curtail any opportunity of an attacker switching a legitimate file with a malicious one.

VI. CONCLUSION

Motivated by the vulnerabilities found in some of the algorithms, hashing will continue to evolve. As this evolution has been a cat and mouse game for many years, it will continue to be so for many more years to come. As a new algorithm is released, so will the next powerful computer to crack it. As a new compression function is created, a new methodology will be found to crack it. In the midst of this never ending churn, there are ways that you can reduce your opportunity of risk exposure – choosing long passwords, downloading data only from trusted sources, and using the strongest hashing or encryption solution available to you. By

taking these steps, you will hopefully redirect the attention of any hacker or criminal to those who were not as cautious.

REFERENCES

- *[1] B. Schneier, *Applied Cryptography*, New York, New York: Wiley Publishing, 1996, pp.30-31,428-459.
- *[2] “Cryptographic Hash Function”, Retrieved from http://en.wikipedia.org/wiki/Cryptographic_hash_function
- [3] D. Melber, “Protect Against Weak Authentication Protocols and Passwords”, Retrieved from <http://www.windowsecurity.com/articles/Protect-Weak-Authentication-Protocols-Passwords.html>
- [4] “LM Hash”, Retrieved from http://en.wikipedia.org/wiki/LM_hash
- *[5] “Cyclic Redundancy Check”, Retrieved from <http://en.wikipedia.org/wiki/CRC-32>
- *[6] “MD5”, Retrieved from <http://en.wikipedia.org/wiki/Md5>
- *[7] “SHA hash functions”, Retrieved from http://en.wikipedia.org/wiki/SHA_hash_functions
- *[8] S. Bellovin and E. Rescorla, “Deploying a New Hash Algorithm”, Retrieved from <http://www.cs.columbia.edu/~smb/papers/new-hash.pdf>
- *[9] H. Dobbertin, “The Status of MD5 After a Recent Attack”, *CryptoBytes*, Volume 2, Number 2, Summer 1996, Retrieved from <ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto2n2.pdf>
- [10] “Birthday Attack”, Retrieved from http://en.wikipedia.org/wiki/Birthday_attack
- *[11] S. Ross, *A First Course in Probability*, New Jersey: Prentice Hall, 1994, pp. 42.
- [12] “NIST Comments on Cryptanalytic Attacks on SHA-1”, Retrieved from <http://csrc.nist.gov/CryptoToolkit/shs/NISTHashComments-final.pdf>
- *[13] X. Wang and H. Yu, “How to Break MD5 and Other Hash Functions”, Retrieved from <http://www.infosec.sdu.edu.cn/paper/md5-attack.pdf>
- *[14] X. Wang et al., “Efficient Collision Search Attacks on SHA-0”, Retrieved from <http://www.cs.cmu.edu/~dbrumley/srg/spring06/sha-0.pdf>
- [15] B. Schneier, “Cryptanalysis of MD5 and SHA: Time for a New Standard”, Retrieved from <http://www.schneier.com/essay-074.html>
- [16] Z. Shuanglei, “Parameter optimization f time-memory trade-off cryptanalysis in RainbowCrack”, Retrieved from <http://www.antsight.com/zsl/rainbowcrack/optimization/optimization.htm>
- *[17] P. Oechslin, “Making a Faster Cryptanalytic Time-Memory Trade-Off”, Retrieved from <http://lasecwww.epfl.ch/~oeechslin/publications/crypto03.pdf>
- [18] “How to prevent Windows from storing a LAN manager hash of your password in Active Directory and local SAM databases”, Retrieved from <http://support.microsoft.com/kb/299656/>
- [19] “Key strengthening”, Retrieved from http://en.wikipedia.org/wiki/Key_strengthening
- [20] “Salt (cryptography)”, Retrieved from http://en.wikipedia.org/wiki/Salt_%28cryptography%29