GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Security Essentials (Security 401)"
at http://www.giac.org/registration/gsec

A Guide to Hash Algorithms

Britt Savage
GIAC GSEC Practical Assignment
Version 1.4b
April 18, 2003

© SANS Institute 2003, Author retains full rights.

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

Abstract

Hash algorithms are one-way functions used to insure message, file, and password integrity on local systems or while passing information across networks. There are many algorithms out there to choose from when trying to decide which is best for any specific application. However, a developer must look at two primary factors. First he must look at how secure he needs to keep the data. Then, he must also look at the response time needed. If the developer is trying to make sure that the correct message sent via an instant messenger is received, an algorithm as simple as CRC-32 might be the right way to go if speed is more important than security. On the other hand, if he is trying to store passwords on a critical system, a more secure algorithm is definitely needed. Although much slower, REPIMD-160 and SHA provide a lot more security.

In this paper, I will describe the uses of hash algorithms and the factors that separate one hash algorithm from the next. I will also go into some detail about how MD2, MD4, MD5, HAVAL, SHA (all versions), REPIMD-160, and Tiger work, along with their strengths and some problems with them. I have also used Crypto++™ Library 5.1 [11] to evaluate the running times of these and many more hash algorithms.

Uses of Hash Algorithms

Hash algorithms are used in cryptography to detect tampering. The first way that hash algorithms are used is to verify passwords. They are also used to verify that a message that is received is the same message that was originally sent. There are other uses, but these are the ones that I will concentrate on in this paper.

Passwords

A problem with using passwords on a system is storing the passwords so that they cannot be read by someone with access to the filing system. If a password is stored on the system in plain text, anyone that could get into the filing system would then be able to just read the password and use it to access an account. An obvious way around this is to somehow store the password so that it is not in plain text. This is where hash algorithms come in. When a user creates a password, it is then hashed using a secure hash algorithm. Now, only the hash value is stored on the system. When the user comes to login, he types in his password, and the system will then hash his password and compare it to the one that it has on file. If these two hash values are equal, then, the system knows that the password is correct. If someone with access to the filing system could read the hash value that is stored on it, he would have to either "un-hash" the value or be able to type in a password that hashes to the same value. We will get to that later when we discuss the properties of a good hash algorithm.

© SANS Institute 2003, Author retains full rights.

Message Authentication

In verifying that a message is authentic, a message is sent over a network along with its hash value. When the receiver gets the message and its hash value, it can run the same hash algorithm on the message and see that the output is the same as the hash value that was sent. The following example shows how this would be done.

1) A sender hashes a message, m, getting h(m), the digest.
2) The sender then sends m and h(m) to the receiver.
3) The receiver hashes m with the same hash algorithm to get h'(m).
4) If h(m) = h'(m), then the receiver knows that the message is authentic.

The problem with this is that it is very vulnerable to a man in the middle attack. Suppose someone intercepted the message and the hash value. He could then change the message, create a new hash value and send them both on to the receiver. When the receiver hashes the new message, he will see that it equals the hash value that he has received and will think that the message is authentic, when, in reality, it is not.

To get around this type of problem the sender can first encrypt the message, but this can be a very slow process. Instead, the sender can just encrypt the hash value, if he is not worried about keeping the message itself secret. This allows the sender to just encrypt the hash value instead of the entire message. The following example shows how a message could be sent with its hash value encrypted using a public key encryption technique.

1) A sender hashes a message, m, getting h(m).
2) The sender then encrypts h(m) with his private key, k1, getting e(k1, h(m)), which is the digital signature of the message.
3) Now the sender will send m and e(k1, h(m)) instead of m and h(m).
4) When the receiver gets the message, he will now un-encrypt the digital signature, e(k1, h(m)), with the sender's public key to get back h(m).
5) The receiver will hash the message, with the same hash algorithm used by the sender, to get h'(m).
6) If h(m) = h'(m), then the receiver will know that the message is authentic.

Now if a man in the middle were to intercept the message and the digital signature, he would have to be able to create a message that hashes to the same digital signature or be able to create another digital signature that could be correctly decrypted with the public key.

Definition of Hash Algorithms

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

Now that we know what hash algorithms are used for, what exactly makes an algorithm a hash algorithm? Hash algorithms are defined as a function h, with a minimum of the following properties [9]:

- h maps a finite input of length x to a fixed output, h(x) of length n
- h(x) is easy to compute.

The first requirement means that any given hash algorithm will produce an output of the same length for every message that it hashes. MD5, for example, creates a 128-bit hash value, no matter the length of the message. The second requirement is relatively self-explanatory.

Although these properties are what define a hash algorithm, an algorithm that only adheres to them is not very useful when it comes to cryptography. The following properties are the ones that make a hashing algorithm useful in cryptography [9]:

- Preimage resistance: Given h(x) it is difficult to compute x
- 2^{nd} Preimage resistance: Given x, it is difficult to find x' such that h(x) = h(x')
- Collision: it is difficult to find x and x' such that h(x) = h(x')

Preimage resistance keeps the content of the message that was hashed secret. In our password example above, if the person with access to the filing system could compute the password from the hash of the password, then what good would it do to even bother with hashing?

The same is true for 2^{nd} preimage resistance. If someone knew what the hash value was and could compute another password that hashed to the same value, they would not even have to know what the original password was. This is also true when a hash algorithm is used to verify the integrity of a message. If the man in the middle could craft a message that hashed to the same value as the original message, then the receiver would never know that the message was ever tampered with.

The third requirement is in place to keep an adversary from doing a "birthday attack." A "birthday attack" is based on probability. If there were 23 people in a room, there is only a 23 in 365 (less than 1 in 15) chance that one of them has the same birthday as you. Surprisingly, though, there is a greater than 1 in 2 chance that any two of these people have the same birthday. In mathematical terms, out of 2^n possible choices, by selecting $2^{n/2}$ choices at random with replacement, there is a 1 in 2 chance of two of your choices being the same [4].

How does this apply to hash algorithms? An adversary could select a message that a certain sender is likely to sign, an innocuous message, and one that the adversary wants to send, a target message. If the sender is going to use a hash

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

algorithm that creates a hash value of n bits, then the adversary would create $2^{n/2}$ versions of each of the two messages. This gives him a 1 in 2 chance that a version of the innocuous message and a version of the target message will hash to the same value. The adversary would then get the sender to digitally sign the innocuous message and then replace it with the target message. Since they both have the same hash value, then he can simply use the digital signature that he has obtained.

You might think that this would mean that the adversary would have a hard time coming up with so many messages without changing the meaning of the message that he wanted to send, but in reality this would be quite easy. He could simply insert invisible characters, such as a space at the end of a line, in the message to craft the message so that it hashed to the correct value. In a 32-line message, the adversary could possibly create four billion different messages just by adding one space at the end of different lines. This is enough to make a hash algorithm that creates an output of 2^{64} bits in length susceptible to a "birthday attack."

Hash Algorithms

MD2

The MD2 algorithm was developed in 1989 by Ronald Rivest for use on older 8-bit machines. It was the first one-way hash function developed in the line developed by Rivest. It produces a 128-bit digest from a message of any length, which is padded to make its total length divisible by 16. Its implementation is specified in RFC1320.

The algorithm for MD2 is as follows [8]:

1) Pad the message to make it 16*n bits in length, where n is an integer.
2) Append the 16-bit checksum as defined in RFC1319.
3) Resulting message is processed in 16-bit blocks.

It has been shown that collisions can be constructed if checksum is left off and that the result of the intermediate compressions can also yield collisions. This does not mean, though, that a collision can be found in the final output of the algorithm. Due to these facts, it is not recommended to use MD2. It is feared that the implementation has to be too precise to make this a good practical hashing algorithm. MD2 is also much slower and less efficient than the much newer MD4 and MD5. This is mostly due to the fact that it was developed for 8-bit machines.

MD4

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

The MD4 algorithm was developed in 1990 for 32-bit machines by Rivest as a replacement for the MD2 algorithm. It also produces a 128-bit digest, but it pads arbitrary length messages so that the final length plus 448 is divisible by 512. Its implementation is specified in RFC1320.

A shortened version of the algorithm is as follows [5]:

1) Message is padded to length n, where n + 448 is divisible by 512.
2) 64-bit binary representation of message is appended to message.
3) Message is processed in 512-bit blocks using the Damgård/Merkle iterative structure.
4) Each block is processed in 3 rounds.

Boer/Bosselaers has shown an attack on the first and second round of the algorithm and Merkle has also shown an attack on the second and third rounds. Attacks have been shown if the first or last round is missing in the implementation as well. Collision can now be found on the entire algorithm in less than a minute on a relatively slow PC. Although MD4 is a very fast and efficient algorithm, MD4 is now considered to be broken [11].

MD5

In 1991 Rivest developed MD5 as an upgrade to MD4. It is different than MD4 in that it processes each block in 4 more complex rounds, which are different than the 3 round that MD3 uses. It is also slower than MD4 but much more secure. The algorithm is defined in RFC1321.

The algorithm is as follows [5]:

1) Pad message so its length plus 448 is divisible by 512.
2) Append a 64-bit value to the end of the message
3) Initialize the 4-word (128-bit) buffer (A,B,C,D).
4) Process the message in 16-word (512-bit) blocks:
    a. using 4 rounds of 16 bit operations each on the chunk & buffer
    b. adding the output to the input to form the new buffer value
5) Output hash value is the final buffer value

Each round is computed by [5]:

1) each round has 16 steps:
2) $R(a,b,c,d,M_i,s,t_i)$: $a = b + ((a+F(b,c,d)+M_i+t_i)<<s)$
3) where $F(b,c,d)$ is a different nonlinear function in each round
4) t_i is a value derived from sine

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

Pseudo-collisions have been found for MD5. Pseudo-collisions are collisions in the compression function in the hash algorithm, but this does not necessarily mean that there will be a collision in the hash function itself. There have been a small number of other collisions found as well. In 1996, Dobbertin announced that collisions for the compression function could be found on a PC at the time in less than 10 hours. It is thought to be a very difficult task, though. Another frightening thought about MD5 has been brought up. It is reported that a machine costing a few million dollars could be built that could find a collision on the entire algorithm in a matter of a few weeks [6]. Despite this, MD5 is still considered secure for most applications where a very high degree of security is not required.

HAVAL

HAVAL (Hash of Variable Length) was released in 1992 by Y. Zheng, J. Pieprzyk and J. Seberry of the University of Wollongong, Australia. Today, it is used in Tripwire. It is different than the MD family in that it can produce digests of 128, 160, 192, 224, and 256 bits in length.

Keys to the HAVAL algorithm [5]:

- It Processes messages in 1024-bit blocks.
- It uses an 8-word buffer and 3 to 5 rounds of 16 steps each.
- It uses highly non-linear 7-variable functions in each step.

HAVAL can be considered superior to MD5 in many ways. First of all, it can produce digests of up to 256 bits in length compared to 128 bits in length. In a brute force "birthday attack" to produce a collision, what would take 2^{64} attempt to get a 1 in 2 chance of finding a collision in MD5 would take 2^{128} attempts to get the same probability. Also, HAVAL is much faster than MD5, and no one has yet shown how to produce a collision in any part of HAVAL. This, in my opinion, may be because HAVAL has not been subject to the same type of cryptanalysis that MD5 has undergone.

SHA

SHA was introduced by NIST & NSA in 1993 and was revised in 1995 after some concerns, which were never released, were found by NIST. It produces a 160-bit digest. Today it is in use as the standard for the DSA signature scheme. It, like RIPEMD-160, is based on the MD5 algorithm and was designed to replace it.

Its algorithm is as follows [5]:

1. pad message so its length is a multiple of 512 bits
2. initialise the 5-word (160-bit) buffer (A,B,C,D,E) to
3. (67452301,efcdab89,98badcfe,10325476,c3d2e1f0)
4. process the message in 16-word (512-bit) chunks

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

5. expand the 16 words into 80 words by mixing and shifting
6. using 4 rounds of 20 bit operations each on the chunk & buffer
7. $(A,B,C,D,E) <- ((E+F(t,B,C,D)+(A<<5)+W_t+K_t),A,(B<<30),C,D)$
8. adding the output to the input to form the new buffer value
9. output hash value is the final buffer value

Although slower, SHA is more secure than MD5 due to a variety of reasons. First, it produces a larger digest, 160-bit compared to 128-bit, so a brute force attack would be much more difficult to carry out. Also, no known collisions have been found for SHA. As I have mentioned, it is claimed that a collision can be found in MD5 in a relatively short period of time.

Since the first introduction of SHA, newer versions have been introduced that are much more secure than the original. They are:

- SHA-256 -- produces a digest of 32 bytes
- SHA-384 -- produces a digest of 48 bytes
- SHA-512 -- produces a digest of 64 bytes

These three were all introduced in 2000. The SHA-256 algorithm uses a 512-bit block, like SHA-1, while SHA-384 and SHA-512 use 1024-bit blocks. Another key difference is that SHA-1 uses 4 rounds where SHA-256 uses 64 and SHA-386 and SHA-512 both use 80 [3]. This makes the cryptanalysis much more difficult. So far, there have been no known successful attacks on the newer versions of SHA. They can be considered among the most secure, although they are not very fast. SHA is now used in the Digital Signature Algorithm, which is the US federal signature scheme [7].

RIPEMD-160

RIPEMD-160 was developed in Europe by researchers whom were involved in trying to break MD5 and MD4. They came up with an algorithm that is very similar to these two but has a few differences. First of all, it uses two parallel lines of 5 rounds of 16 steps [5]. It produces a 160-bit digest, which is much more secure than MD5, and it is thought to be even more secure than SHA. It is slower than the two but is more preferable in a situation where security is the biggest concern.

Tiger

Ross Anderson and Eli Biham developed tiger in 1996. They were interested in creating a secure hash algorithm that could run efficiently on new 64-bit machines, as well as 32-bit machines. Also, at the time, MD4 was being broken and collisions for Snerfu were being found. They believed that the weaknesses in these algorithms would lead to MD5, Snerfu-8, and other algorithms based on these to be broken in the near future [1].

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

Anderson and Biham came up with an algorithm that produced a 196-bit digest from 3 nonlinear rounds. So far, there have been no known attacks on Tiger, and can be considered secure. Tiger is much faster than SHA, but has a smaller digest than the newer versions of SHA, such as SHA-256 and SHA-512. Therefore, Tiger can be considered a bit less secure than these newer versions. Tiger is slower than MD5, but can be considered more secure.

Comparison of Hash Algorithm Speed

In evaluating the run time of the different hash algorithms, I used Crypto++™ Library 5.1 [7], which is a freeware library of individual implementations of various hash algorithms. It was run on a PC with a Pentium 3 processor running at 800 MHz with 256 MB of RAM. The OS was Windows 2000. I have included the results of some other hash functions that I have not discussed in this paper in order to give the reader a better idea of the many choices that are available when selecting a hash algorithm. Each test was run 5 times and the average of each test is shown. The results are as follows:

Algorithm	Bytes Processed	Time Taken	Megabytes(2^20 bytes)/Second
CRC-32	67108864	0.961	66.597
Adler-32	67108864	0.921	69.490
MD2	262144	0.701	0.357
MD5	33554432	1.262	25.357
SHA-1	8388608	1.192	6.711
SHA-256	4194304	1.292	3.096
SHA-512	2097152	0.731	2.736
HAVAL (pass=3)	8388608	0.971	8.239
HAVAL (pass=4)	4194304	0.751	5.326
HAVAL (pass=5)	4194304	0.962	4.158
Tiger	16777216	1.191	13.434
RIPE-MD160	4194304	0.711	5.626
Panama Hash (little endian)	8388608	0.701	11.412
Panama Hash (big endian)	8388608	0.992	8.065

Note: These results may not be completely accurate since they are subject to each individual implementation, of which some may be better tuned for the specific machine they were run on. Also, there may have been some

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

background interference where another application, such as the OS, could have been using resources.

As you can see, other than a few insecure algorithms, MD5 wins the throughput contest by a long shot. This makes it an ideal candidate for a situation in which security is important but speed is the real factor. An example of this would be something like non-critical email. But, as I have discussed earlier, when security is the more important factor, it is better to use either SHA or RIPEMD-160. These two, especially RIPEMD, are much more secure and should be used for situations like storing passwords on a critical system.

Summary

If the world was a perfect place and there was a perfect hash algorithm that ran very fast and could not be broken in a million years, no one would have to decide which hash algorithm to use. But, this is not the case. There are two main factors that a developer has to weigh in order to determine the most appropriate algorithm for his particular case. When it comes to more speed and less security, an algorithm like MD5 might be the best bet. It provides a moderate amount of security but can hash data very quickly. On the other hand, if security is the most important factor, then one would be better off with SHA-512 or REPIMD-160. They are much slower than MD5, but provide more security due to a larger digest and fewer security holes. There are many algorithms that are faster, slower, more secure, and less secure than these, and in the end the developer must decide which is best for his particular implementation.

© SANS Institute 2003, As part of GIAC practical repository. Author retains full rights.

Bibliography

1. Anderson, Ross & Biham, Eli, Tiger, A Fast New Hash Function, http://www.cs.technion.ac.il/~biham/Reports/Tiger/tiger/tiger.html

2. "Authentication, Hash Functions, and Digital Signatures", http://islab.oregonstate.edu/documents/People/stallings/6.pdf.

3. Barreto, Paulo S. L. M., "The Hashing Function Lounge, http://planeta.terra.com.br/informatica/paulobarreto/hflounge.html (2003, March 4)

4. Crypto FAQ, http://www.x5.net/faqs/crypto/q96.html, http://www.x5.net/faqs/crypto/q99.html.

5. "Cryptography - Lecture 18 - Authentication, Hash Algorithms", http://www.kewlstuff.co.za/cryptolessons/Lecture%2018.htm.

6. "Cryptographic Algorithms" http://starship.python.net/ssh/java/Cryptographic_algorithms.html.

7. Dai, Wei, "Crypto++™ Library 5.1", http://www.eskimo.com/~weidai/cryptlib.html (2003, March 20).

8. Kaliski, B., "The MD2 Message-Digest Algorithm" http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1319.html (1992, April).

9. Perrig, Adrian, "Requirements for Traditional Hash Functions", http://paris.cs.berkeley.edu/~perrig/projects/validation/node3.html#SECTION00021000000000000000, (1999, Sept. 15).

10. Persits Software, Inc., Crypto 101, "One-way Hash Functions", http://www.aspencrypt.com/crypto101_hash.html (2000).

11. Robshaw, M. J. B., RSA Laboratories' Bulletin, http://www.securitytechnet.com/resource/crypto/algorithm/Symmetric/RSAbulletn4.pdf, (1996, Nov. 12).

© SANS Institute 2003, Author retains full rights.