Home

Posted: November 12, 2007

How Hash Algorithms Work

This page was written for people who *really* want to know exactly how hash algorithms work. The following is a step-by-step walk through of exactly how hash algorithms work. This is written mainly for people with very good knowledge about computers, encryption, and logical operators. I did try to write it so that everyone could understand it, but you might find it boring/dry/confusing if you aren't really interested in how hash algorithms work and know a fair amount already.

What is a 'Hash Algorithm'?

A hash function is simply an algorithm that takes a string of any length and reduces it to a unique fixed length string.

What are hash algorithms used for?

Hashes are used to ensure data and message integrity, password validity, and are the basis of many other cryptographic systems.

Important properties:

Each hash is unique but always repeatable

The word 'cat' will hash to something that *no* other word hashes too, but it will *always* hash to the same thing.

The function is 'one way'.

If you are given the value of what 'cat' hashes too but you didn't know what made it, you would never be able to find out that 'cat' was the original word.

There are many different hash functions but the one I will be concentrating on today is called the Secure Hash Algorithm 1 or SHA-1.

Example:

'test' => SHA-1 => 'a94a8fe5ccb19ba61c4c0873d391e987982fbbd3'

Every time that anyone anywhere runs the word 'test' through the SHA-1 function, they should always

get: a94a8fe5ccb19ba61c4c0873d391e987982fbbd3.
Also, it should be computationally infeasible to find any other word which also hashes to a94a8fe5ccb19ba61c4c0873d391e987982fbbd3.
Finally, if I were to give you only 'a94a8fe5ccb19ba61c4c0873d391e987982fbbd3' and tell you that it came from the SHA-1, you should have absolutely no way to figure out what was put into the function to create that.

Almost all computer passwords are stored in this fashion (though hopefully not with SHA-1). When you create a password the computer runs it through a hash function then stores only the result. Because the function is 'one-way', even if someone were to gain access to the file that stores that output, they shouldn't be able to figure out your password.
When the computer prompts you to enter your password to log in, it will simply hash whatever you give it and then compare it to the stored hash of your password.
This is often why passwords must be reset. Because the computer never stores your actual 'plain text' password and only a fingerprint of it, there is no way for it to tell you what your password was.

How hash algorithms actually work:

All of the information above could easily be found elsewhere on the internet in a much more thorough and accurate way. In fact, I have intentionally left out many details because there weren't relevant to my main cause. The reason I decided to write this was that I was absolutely fascinated by how these functions actually work but was completely unable to find a working example online. Wikipedia not only has a great article, but also very wonderful 'pseudocode'. Pseudocode is like the blueprints without any of the details. What I was unable to find was anything that would actually show you step by step exactly how a word gets hashed. It's my plan to walk you through from start to finish exactly what happens inside a hash function. If you're more interested in what a hash function is, what they are used for, or a general overview of how they work, I highly suggest you read this wiki on the subject. No matter what your intent or current knowledge, you should read this post if you haven't done so already. If you want to see a working example of how a string becomes a hash, then please read on.

Step 0: Initialize some variables

There are five variables that need to be initialized.

h0 = 01100111010001010010001100000001

h1 = 11101111110011011010101110001001

h2 = 10011000101110101101110011111110

h3 = 00010000001100100101010001110110

h4 = 11000011110100101110000111110000

Step 1: Pick a string

In this example I am going to use the string: 'A Test'.

A Test

Step 2: Break it into characters

A		T	e	s	t

Note that spaces count as characters.

Step 3: Convert characters to ASCII codes

Each character should now be converted from text to ASCII. ASCII or the 'American Standard Code for Information Interchange' is a standard that allows computers to communicate different symbols by assigning each one a number. No matter what font or language you use, the same character will always have the same ASCII code.

65	32	84	101	115	116

Note that 'T' and 't' are different ASCII characters.

Step 4: Convert numbers into binary

Binary is simply base two. All base ten numbers are now converted into 8-bit binary numbers. The eight-bit part means that if they don't actually take up a full eight place values, simply append zeros to the beginning so that they do.

01000001 00100000 01010100 01100101 01110011 01110100

Step 5: Add '1' to the end

Put the numbers together:

010000010010000001010100011001010111001101110100

Add the number '1' to the end:

0100000100100000010101000110010101110011011101001

Step 6: Append '0's' to the end

In this step you add zeros to the end until the length of the message is congruent to 448 mod 512. That means that after dividing the message length by 512, the remainder will be 448. In this case, the length of the original message in binary is 48 + 1 from the last step. That means that we will need to add a total of 399 zero's to the end.

0100000100100000010101000110010101110011011101001000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000

If your original message were 56 characters long, it would be exactly 448 digits long once converted into binary. After adding the '1' to the end of that, the new message would be 449 characters long. If that were the case you would need to append 511 zero's to make the message congruent to 448%512.

If your original message were 64 characters long, it would be exactly 512 digits long once converted into binary. After adding the '1' to the end of that, the new message would be 513 characters long. If that were the case you would need to append 447 zero's to make the message congruent to 448%512.

Step 6.1: Append original message length

This is the last of the 'message padding' steps. You will now add the 64-bit representation of the original message length, in binary, to the end of the current message.

In this case our original message was 48 characters long.

48 in binary is expressed as: 110000

However, since the number must be 64-bits or digits long we must now add 58 zero's to the beginning of that number prior to adding it to the current message.

```
01000001001000000101010001100101011100110111010010000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000110000
```

The message length should now be an exact multiple of 512.

Step 7: 'Chunk' the message

We will now break the message up into 512-bit chunks. In this case the message is only 512 bit's long, so there will be only one chunk that will look exactly the same as the last step.

```
01000001001000000101010001100101011100110111010010000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000110000
```

Step 8: Break the 'Chunk' into 'Words'

Break each chunk up into sixteen 32-bit words

```
0: 01000001001000000101010001100101
```

1: 01110011011110100100000000000000

2: 00000000000000000000000000000000

3: 00000000000000000000000000000000

4: 00000000000000000000000000000000

5: 00000000000000000000000000000000

6: 00000000000000000000000000000000

7: 00000000000000000000000000000000

8: 00000000000000000000000000000000

9: 00000000000000000000000000000000

10: 00000000000000000000000000000000

11: 00000000000000000000000000000000

12: 00000000000000000000000000000000

13: 00000000000000000000000000000000

14: 00000000000000000000000000000000

15: 00000000000000000000000000110000

Step 9: 'Extend' into 80 words

This is the first sub-step. Each chunk will be put through a little function that will create 80 words from the 16 current ones.

This step is a loop. What that means is that every step after this will be repeated until a certain condition is true.

In this case we will start by setting the variable 'i' equal to 16. After each run through of the loop we will add 1 to 'i' until 'i' is equal to 79.

Step 9.1: XOR

We begin by selecting four of the current words. The ones we want are: [i-3], [i-8], [i-14] and [i-16].

That means for the first time through the loop we want the words numbered: 13, 8, 2 and 0.

0: 01000001001000000101010001100101

2: 00000000000000000000000000000000

8: 00000000000000000000000000000000

13: 00000000000000000000000000000000

The next time through the loop we want words: 14, 9, 3 and 1.

After the fifth time through the loop we will want words: 17, 12, 6 and 4.

Note that word 17 doesn't exist yet, but it will after the first run of the loop. That means that after seventeen passes through the loop we will be using entirely words that aren't part of the original sixteen.

Now that we have our words selected we will start by performing what's known as an 'XOR' or 'Exclusive OR' on them. In the end all four words will be XOR'ed together, but you can think of it as first doing [i-3]XOR[i-8] then XOR'ing that by [i-14] and that again by [i-16]. XOR is one of a few simple logical operators. All it means is that you compare the two numbers bit by bit and if exactly one of them has the value '1', output a '1'. However, if both numbers have a '0' for that bit, or they both have a '1', output a '0'. This works very similarly to a logical 'OR' which will be used later. The only difference is that an 'OR' will return a '1' so long as either column has a '1' even if both of them do.

Let's see how it works:

:	1011010101
:	0101001011
example XOR	
Out:	1110011110

So for our example we begin by XOR'ing:

13:	00000000000000000000000000000000
8:	00000000000000000000000000000000
13 XOR 8	
Out:	00000000000000000000000000000000

Now we XOR that with word number [i-14]:

:	00000000000000000000000000000000
2:	00000000000000000000000000000000
(13 XOR 8) XOR 2	
Out:	00000000000000000000000000000000

Now we XOR that with word number [i-16]:

:	00000000000000000000000000000000
0:	01000001001000000101010001100101
((13 XOR 8) XOR 2) XOR 0	
Out:	01000001001000000101010001100101

Step 9.2: Left rotate

Perform a carry through left bit rotation by a factor of one. This is very simple. All you do is remove the first digit on the left and append it to the end. This effectively shifts the number over to the left by one. Here's how it looks:

Output from the last step:

01000001001000000101010001100101

Left Rotate 1:

10000010010000001010100011001010

Once you are done with that, you can store the variable as a new word. In this case it will be word number 16 *(keep in mind that we start counting at 0)*.

After step nine is complete we will now have 80 words that look like this:

```
0: 01000001001000000101010001100101
1: 01110011011101001000000000000000
2: 00000000000000000000000000000000
3: 00000000000000000000000000000000
4: 00000000000000000000000000000000
5: 00000000000000000000000000000000
6: 00000000000000000000000000000000
7: 00000000000000000000000000000000
8: 00000000000000000000000000000000
9: 00000000000000000000000000000000
10: 00000000000000000000000000000000
11: 00000000000000000000000000000000
12: 00000000000000000000000000000000
13: 00000000000000000000000000000000
14: 00000000000000000000000000000000
15: 00000000000000000000000000110000
16: 10000010010000001010100011001010
17: 11100110111010010000000000000000
18: 00000000000000000000000001100000
19: 00000100100000010101000110010101
20: 11001101110100100000000000000001
21: 00000000000000000000000011000000
22: 00001001000000101010001100101010
23: 10011011101001000000000001100011
24: 00000100100000010101000000010101
25: 11011111110101110100011001010101
```

```
26: 00110111010010000000000000000111
27: 00000000000000000000001100000000
28: 00100100000010101000110010101000
29: 01101110100100000000000111101110
30: 00010110100001000001000111000001
31: 10110010100011110001100111110110
32: 11010000101000111111001010100011
33: 01010110011101100000110000000010
34: 10010000001010100011001100100000
35: 10101000010001010100000111101101
36: 01101101010110000100011100000011
37: 11001010001111000110010011011010
38: 01100110100001010100011000100111
39: 00110111010010000011000110000111
40: 01010010101011011000110011010110
41: 11011110010010000000111011100001
42: 01101000010000010001110000010001
43: 00101000111100011001111110101011
44: 00000011001111011000100100010111
45: 11111100110001001100000110100110
46: 00010000101001100111010111011101
47: 10100001000110010101101011001001
48: 11011101110000010001100111100111
49: 01100001101110100100110110100110
50: 01101000010101000110010111110110
51: 00101110100100110100011011110111
52: 11010010101101011000101100101010
53: 11010011110010011110001000011010
54: 00010100001110111111001110110110
55: 00110101010110011111110100001011
56: 01101001110010001101011001110100
57: 00000110011100000111111010110101
58: 01101100111000100001101111110110
59: 00100110110111011001110100011101
60: 10001110101111000001001010101011
61: 11000101111011001100010100000111
62: 11111111000000100000010100100011
```

```
63: 11110110100011011111000110011110
64: 00110011011000101101111011000100
65: 01101100101101101110000110001111
66: 01000001000111000000100101101000
67: 11010001110010111100111001101001
68: 01001001000010010001011101110000
69: 11000100110000011010011011111100
70: 10100110011101011101110100010000
71: 00011001010110101100101010100001
72: 11100101000100110110101101110101
73: 11010100110111011011111001101111
74: 01110100001100011001010100101001
75: 10101111110100111111101000001101
76: 11011000110101010111110100101111
77: 00000111001000100000111010011001
78: 10001011100011011111100111110101
79: 10110111011010010100111100111110
```

Step 10: Initialize some variables

Set the letters A-->E equal to the variables h0-->h4.

A = h0

B = h1

C = h2

D = h3

E = h4

Step 11: The main loop

This loop will be run once for each word in succession.

Step 11.1: Four choices

Depending on what number word is being input, one of four functions will be run on it.

Words 0-19 go to function 1.

Words 20-39 go to function 2

Words 40-59 go to function 3

Words 60-79 go to function 4

Function 1

Remember that 'OR' function I mentioned earlier, we're going to be using that and a logical 'AND' for this function.

Just to refresh: A logical 'OR' will output a '1' if either **or** both of the inputs are '1'.

A logical 'AND' will output a '1' if the first input **and** the other is a '1'.

For all logical operations a '0' will be output if the conditions are not met. The only other logical operator we will be using is called a 'NOT'. A logical not only takes one input and outputs the opposite. If you put in a '1' you will get a '0', if you put in a '0' you will get a '1'. A logical 'NOT' is often represented as an exclamation point(!).

The first step of function 1 is to set the variable 'f' equal to: (B AND C) OR (!B AND D)

B:	11101111110011011010101110001001
C:	10011000101110101101110011111110
B AND C	
Out:	10001000100010001000100010001000

!B:	00010000001100100101010001110110
D:	00010000001100100101010001110110
!B AND D	
Out:	00010000001100100101010001110110

B AND C:	10001000100010001000100010001000
!B AND D:	00010000001100100101010001110110
(B AND C) OR (!B AND D)	
F:	10011000101110101101110011111110

The second step of function 1 is to set the variable 'k' equal to:
01011010100000100111100110011001

Function 2

For this function we will be using the 'XOR' operation exclusively.
Just to refresh: A logical 'XOR' will output a '1' if either the first **or** the second of the inputs are '1' *but* **not** both.

The first step of function 2 is to set the variable 'f' equal to: B XOR C XOR D
Of course by this time our variables have changed. Here's what this step will actually look like by the time we get to it:

B:	11011110010001000100111100111010 1
C:	10110101000101000100100011110000
B XOR C	
Out:	01101001100111001101011010000101

B XOR C:	01101001100111001101011010000101
D:	01001110101011001011101010110111
(B XOR C) XOR D	
F:	00100111001100000110110000110010

The second step of function 2 is to set the variable 'k' equal to:
01101110110110011110101110100001

Function 3
For this function we will be using the 'AND' and 'OR' operations.
The first step of function 3 is to set the variable 'f' equal to: (B AND C) OR (B AND D) OR (C AND D)
By this time our variables have changed again. Here's what this step will actually look like by the time we get to it:

B:	01000100110000000111111001110111
C:	00011010100110110011101010111011
B AND C	
Out:	00000000100000000011101000110011

B:	01000100110000000111111001110111
D:	01010011011001010110101011100100
B AND D	

Out:	01000000010000000110101001100100

C:	00011010100110110011101010111011
D:	01010011011001010110101011100100
C AND D	
Out:	00010010000000010010101010100000

B AND C:	00000000100000000011101000110011
B AND D:	01000000010000000110101001100100
(B AND C) OR (B AND D)	
Out:	01000000110000000111101001110111

(B AND C) OR (B AND D):	01000000110000000111101001110111
C AND D:	00010010000000010010101010100000
((B AND C) OR (B AND D)) OR (C AND D)	
F:	01010010110000010111101011110111

The second step of function 3 is to set the variable 'k' equal to: 10001111000110111011110011011100

Function 4
Function 4 is exactly the same as function 2 except that we will set 'k' equal to 11001010011000101100000111010110.

Step 11.2: Put them together

After completing one of the four functions above, each variable will move on to this step before restarting the loop with the next word. For this step we are going to create a new variable called 'temp' and set it equal to: (A left rotate 5) + F + E + K + (the current word).
Notice that other than the left rotate the only operation we're doing is basic addition. Addition in binary is about as simple as it can be.
We'll use the results from the last word(79) as an example for this step.

A lrot 5:	00110001000100010000101101110100
F:	10001011110000011101111100100001
A lrot 5 + F	

Out:	1101111100110100101110101010010101

Notice that the result of this operation is one bit longer than the two inputs. After each iteration the new word should be one bit longer than the last. Sometimes this will be a necessary carrier bit (like the extra place value you need to represent the result of adding the two single digit numbers 5 and 6 in base 10), and when that's not needed you must simply prepend a 1. For everything to work out properly we will need to truncate that extra bit eventually; however, we do **not** want to do that until the end!

A lrot 5 + F:	1101111100110100101110101010010101
E:	11101001001001111110100110101011
A lrot 5 + F + E	
Out:	1010100101111110101101010001000000

A lrot 5 + F + E:	1010100101111110101101010001000000
K:	11001010011000101100000111010110
A lrot 5 + F + E + K	
Out:	11101110000010111011001011000010110

A lrot 5 + F + E + K:	11101110000010111011001011000010110
Word 79:	10110111011010010100111100111110
A lrot 5 + F + E	
Out:	100000100111110001101110010101010100

Now we need to truncate the result so that the next operations will work smoothly. We will remove as much of the beginning(left) until the number is 32 bits or 'digits' long.

32-bit temp:	00100111110001101110010101010100

The only thing left to do at this point is 're-set' some variables then start the loop over. We will be setting the following variables as such:

E = D

D = C

C = B Left Rotate 30

B = A

A = temp

Step 12: The end

Once the main loop has finished there is very little left to do. All that's left is to set:

h0 = h0 + A

h1 = h1 + B

h2 = h2 + C

h3 = h3 + D

h4 = h4 + E

If these variables are longer than 32 bits they should be truncated.

If the original message took more than one 'chunk' to represent, the result from step 11 for each chunk will be added together here.

For our example the 'h' variables will now have these values:

h0 = 10001111000011000000100001010101

h1 = 10010001010101100011001111100100

h2 = 10100111110111100001100101000110

h3 = 10001011001110000111010011001000

h4 = 10010000000111011111000001000011

Finally the variables are converted into base 16 (hex) and joined together.

8f0c0855 915633e4 a7de1946 8b3874c8 901df043

8f0c0855915633e4a7de19468b3874c8901df043

And that's how you go from a string to a 'hash'!

Would you like to see every single step with your own input?

Just enter whatever you want and hit 'See'.

See

Share / Save