Encryption, Hashing, and Complexity: Oh My!

James L. Antonakos
WhiteHat Forensics
Overview

• Encryption, hashing, and complexity are important topics related to information security.
• Encryption is used to provide confidentiality to information (hashing is able to do this to some extent as well).
• Hashing is used to help verify the integrity of information (has it changed?).
• Complexity is associated with the ease or difficulty of cracking an encryption code.
Topics

• Encryption Techniques
• The Caesar Shift
• Transposition Encryption
• Using a Keyword with Substitution Encryption
• A Slight Problem Here
• So Make it Harder
• But Time is Relative
Topics

• A While?
• Key Size
• How many Bits is Enough?
• Now, what about the Technique?
• All right then, but what is Hashing?
• Hashing Examples
• Malware and Hashing
Encryption Techniques

• The foundation of all secure data transmission is an encryption technique. There are many techniques, some better suited for one type of data, others for other types of data.
• But isn’t data just data? No. There is data that represents text, data that represents numbers, data that represents code, etc, and each have characteristics that can be exploited both for compression and encryption.
• Let us take a short tour of several different basic techniques.
Encryption Techniques

- Ciphers
  - Classical
    - Substitution
  - Rotor Machines
  - Modern
    - Private Key
      - Stream
    - Public Key
      - Block
Encryption Techniques

Block

Stream
The Caesar Shift

• This simple encryption technique shifts the alphabet a certain number of letters one way or the other, with wraparound at each end. Here is an example of a Caesar shift using 5 letters of shift:

- ABCDEFGHIJKLMNOPQRSTUVWXYZ
- FGHJKLMNOPQRSTUVWXYZABCDE

• So, an A becomes an F, an E becomes a J, etc, by locating the letter to be encrypted on the first row and writing down the shifted letter from the second row.
The Caesar Shift

• ZSIJWXYFSI?

• To decode, look up each letter of the encrypted message in the second row and write down the decrypted letter from the first row.

• UNDERSTAND?

• Even though this is a simple technique, it is useful and easy to implement in software. It is also easy to share the ‘key’ for the technique... you just need to know how many letters to shift.
Substitution Encryption

• In substitution encryption you just replace each original character with one from its position within the encryption alphabet.

• ABCDEFGHIJKLMNOPQRSTUVWXYZ
• QPWOEIRUTYALSKDJFHGZMXNCBV

• The difficulty here is the entire encryption alphabet must be shared with the receiver.
Transposition Encryption

• This technique does not use an encryption alphabet to transform the letters, but instead rearranges the letters of the message in a specific order, while at the same time making the message unreadable.

• Consider this sample message:

• JOHN LIKES TO EAT HIS MILK AND BREAD

• To help keep track of the blanks between words I am putting in periods. It is not necessary, but will help visualize what is going on.

• JOHN.LIKES.TO.EAT.HIS.MILK.AND.BREAD
Transposition Encryption

• Now we write the letters of the message down in a two-dimensional array (also called a matrix) of letters as shown here, with 6 letters (or periods) on each row of the array.

• JOHN.L
• IKES.T
• O.EAT.
• HIS.MI
• LK.AND
• .BREAD

• The 36 letters (and periods) in the message fill the 6 by 6 matrix of letters exactly. It is a simple matter to add extra blanks (or periods) at the end of the message if the original message is not long enough to fill the matrix.

• Now here comes the transposition part of the technique. We wrote the letters into the matrix one row at a time but we read the letters out one column at a time, as in:

• JIOHL.OK.IKBHEES.RNSA.AE..TMNALT.IDD
Transposition Encryption

• Now, replacing the periods with blank spaces again gives us the encrypted message:

  • JIOHL OK IKBHEES RNSA AE  TMNALT IDD

• Compare the original message with its transposition encrypted counterpart:

  • JOHN LIKES TO EAT HIS MILK AND BREAD
  • JIOHL OK IKBHEES RNSA AE  TMNALT IDD

• As with the other techniques, this method is easy and fast to implement.
Using a Keyword with Substitution Encryption

• To make the sharing of the key easier, you can use a keyword. The keyword must contain all unique letters, no repeats allowed. Here are some sample keywords, and their resulting second rows. Do you see how the second rows are filled in?

• ABCDEFGHIJKLMNOPQRSTUVWXYZ
• SUPERBOWLACDFGHIJKMNQTVXYZ
• ABCDEFGHIJKLMNOPQRSTUVWXYZ
• TRICKYABDEFGHIJKLMNOPQRSTUVWXYZ
• ABCDEFGHIJKLMNOPQRSTUVWXYZ
• ZENITHABCDFGJKLMOPQRSUVWXYZ

• When writing the second row of letters after the keyword, fill in the remaining letters in order. Now you just need to share the keyword to know how to decode the message.
A Slight Problem Here

- Unfortunately, all three of the techniques presented have a limitation: they are easy to crack.
- You can make cracking a message more difficult by using two or more techniques, but the final result will still be crackable within a short period of time.

<table>
<thead>
<tr>
<th>Technique</th>
<th>How to Crack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caesar Shift</td>
<td>Try all possible shift values from 1 to 25 until it works.</td>
</tr>
<tr>
<td>Substitution</td>
<td>Use known rules and patterns. For example, E and I are the most frequent</td>
</tr>
<tr>
<td></td>
<td>letters, so a letter frequency analysis would be a good start. Plus Q is</td>
</tr>
<tr>
<td></td>
<td>typically followed by U, there are known doubles such as LL and OO and EE,</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
</tr>
<tr>
<td>Transposition</td>
<td>Write the message as a matrix one row at a time and read it out one column</td>
</tr>
<tr>
<td></td>
<td>at a time.</td>
</tr>
</tbody>
</table>
So Make It Harder

- Add some bit shifting and the Exclusive-OR operation into the mix and now you’ve made it more difficult to find patterns.
So Make It Harder

- Add the Exclusive-OR operation into the nix and now you’ve made it more difficult to find patterns.
So Make It Harder

- A malware writer used a short XOR decrypting loop at the beginning of the code. The **call $+5** instruction pushes a return address onto to the stack, but this address is the address of the next instruction **pop ecx**. So, these two instructions together give the program *a way to determine the Instruction Pointer*, no matter where in memory the code is loaded and executed. How clever of the malware writer to do this and to encrypt the payload!

```assembly
| Seg000:00000008 | db 49h ; I |
| Seg000:0000000C | db 90h ; E |
| Seg000:0000000D | call $+5 |
| Seg000:00000012 | pop ecx |
| Seg000:00000013 | add ecx, 0Ch |
| Seg000:00000016 | xor byte ptr [ecx], 44h |
| Seg000:00000019 | inc ecx |
| Seg000:0000001A | cmp byte ptr [ecx], 0C3h ; '+' |
| Seg000:0000001D | jnz short near ptr 0FFFFFFD2h |
| Seg000:0000001F | add dword ptr ds:34376288h, 28h ; '(' |
| Seg000:00000026 | add byte ptr ds:8FAC7947h, 44h ; 'D' |
| Seg000:0000002D | inc esp |
| Seg000:0000002E | inc esp |
| Seg000:0000002F | int 83h ; reserved for BASIC |
| Seg000:00000031 | sub al, 0CDh ; '-' |
| Seg000:00000033 | mov ecx, 0AC13E056h |
| Seg000:00000038 | mov edi, 14444444h |
| Seg000:0000003D | sub al, 28h ; '+' |
```

Note however that this address is incorrect and will be 0xFFFFFFFF7h when the XOR takes place.
So Make It Harder

- Note that this technique of encrypting payload codes is one of the techniques used to hide the payload code. Another technique is to rotate the bits in each byte 1, 2, or more places as well. Now, suppose you suspect that the encrypted code contains a URL string somewhere that begins with the characters http. A nice tool called XORsearch will take an input file (the encrypted code in our case) and an input string to search for when trying every combination of XOR values from 0 to FF and every rotation pattern. Here is what XORsearch finds:
But Time is Relative

- It sure is. So, what do I mean by “within a short period of time.” Well, a few seconds is short, so is one minute, a few hours, or even several days or weeks. Why? All may not be short depending on what is being protected by encryption. For a newspaper scramble puzzle, some people may crack it in a few minutes, others in a few hours, while a few may chip away at it for days or weeks before breaking the code.

- But if the data we have encrypted represents an electronic banking transaction, we may not want that message to be cracked for a very long time. But what is a long time? 100 years? 10,000 years? One billion years?

- Ah, we see the difficulties of talking about short or long periods of time. But if it were my own personal encrypted information, I feel that keeping someone waiting for a billion years would be strong enough encryption for me.
But Time is Relative

- So, let’s look at an example of how we can improve the strength of our encryption method. Consider the following group of numbers, which represents the “message” we want to encrypt:

  - 100  23  214  86

- Now, I am going to pick a number from 1 to 16. I am not going to tell you the number. Then I am going to divide each number in the message by my secret number and keep track of the quotient and remainder:

  - 7  2  1  9  15  4  6  2

- Now a switcheroo:

  - 2  7  9  1  4  15  2  6

- and back to just four numbers, using my secret number as a multiplier now, to get the encrypted data:

  - 35  127  71  34
But Time is Relative

• Once again, let’s compare the original data with the encrypted data:

  
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>23</td>
<td>214</td>
<td>86</td>
</tr>
<tr>
<td>35</td>
<td>127</td>
<td>71</td>
<td>34</td>
</tr>
</tbody>
</table>

• Is it possible to see any kind of pattern between the two groups of numbers? Any clue as to my secret number? Hopefully not... but you would agree I think that since my secret number is between 1 and 16 you could try every value until one of them works.

• I will save you the trouble, my secret number is 14. To see how I use 14 to turn 100 into 35, watch this:

  
  • 100 divided by 14 equals 7 with a remainder of 2. Check: 7 times 14 equals 98. 98 plus 2 equals 100.

  • Ok, now I swapped the 7 and 2 to get 2 and 7.

  • Then I multiplied 2 by 14 to get 28 and added 7 to get 35.
But Time is Relative

Well, you might say, this technique is easy to crack too. Just try all the numbers.

- But remember this: Even though I did not tell you my secret number in the beginning, I told you I was dividing the numbers in the original message and also switching their quotient and remainders. So, you knew part of the technique, which helps you while you are attempting to crack it. If I can keep the secret number and the technique secret as well, that is even better because then you will be hard pressed to see a pattern.

- For now, let’s concentrate on the secret number. A range of 1 to 16 is not a very big range and you will have no trouble cracking the code in a relatively short period of time. So, we need to increase the range. How about 1 to 100 trillion? That is much bigger. Think about the poor person who has to try to crack that numeric code, even knowing the multiply/divide switcheroo method? Which number out of 100 trillion choices is the secret number? That will keep the person, or even a computer, busy for a while.
A While?

• Ok, what do I mean by “a while?”

• Computers are very fast, but humans are slow. We do not like to think about things like nanoseconds. Hard to appreciate. But tell us a message takes 5 seconds to crack, and we understand that. Or telling us that a message will take 250 million centuries to crack is also something we can appreciate. Which message is uncrackable? I think you get the idea.

• So, let’s consider a personal computer available today. Maybe you have a dual-core Pentium at 4 GHz (Giga Hertz, 4 billion clock cycles each second). That means its clock period, the time of one clock cycle, is 4 billionths of a second, or 0.25 nanoseconds.

• Now, the Pentium will require from 1 to 4 or more clock cycles to execute an instruction. Let us pretend it takes 2.5 clock cycles, on average, to execute an instruction. And, let us further assume that it takes 16 instructions to perform a decryption on one symbol, if we have the correct secret key. This means we can decrypt one symbol in 10 nanoseconds. That seems a reasonable amount of time and is probably much shorter than what would actually be required, so we are looking at a best-case scenario here.
Now, let us further assume that after 10 nanoseconds and we have decoded one symbol that we can look at that symbol and tell if it is correct. If it is not correct, then we have to choose a new key and try decrypting the same symbol again. In effect, we can now decrypt and check one symbol in 10 nanoseconds. We really can’t do this, but we are pretending.

So, if we can decrypt and check one symbol in 10 nanoseconds, we can check 100 million keys in one second. Each key we try fails and we move on to the next key. Remember, we really can not check 100 million keys in one second with our 4 GHz Pentium because we are pretending.

Ok, I mentioned 100 trillion choices before. Let’s look at that number. 100 trillion keys divided by the ability to check 100 million keys in one second gives 1 million seconds, which works out to 11 days, 13 hours, 46 minutes, and 40 seconds. I said that 100 trillion choices (keys) would keep the person or computer busy “for a while.” Do you agree that over 11 and a half days is “a while?”
A While?

• Would you also agree that, since we have been pretending, that it will actually take longer than 11 and a half days to crack the code? How much longer does not matter, even if it is 10 times longer or 100 times longer, because two things are true:

• There is a limit to how far off my estimate was. Maybe it would take 175 times longer, but I assure you it will not take 500 times longer, or 5000 times longer.
• Some people are willing to wait. So, even if it takes a month, or a year, or maybe a lifetime, we really need to think hard about what a long time is.

• Because of these two reasons, we can not get excited about breaking codes just because a 5 GHz Pentium, or a 50 GHz Pentium, or even a 5000 GHz Pentium rolls out. We can easily choose a key that is so large that even the fastest computers, or even cluster of computers such as a supercomputer, will not be able to break the code.
Key Size

• It is useful to represent the size of a secret number key by its bit size. The more bits there are in a key, the higher the range of numbers that can be represented. Look at the table to get an idea.

• The equation is simple:

\[ R = 2^K - 1 \]

• where \( K \) is the number of bits in the key and \( R \) is the highest number in the range.

• Remember our 100 trillion choices from before? How many bits are there in that key? From the table we can see that the key size falls somewhere between 32 and 64 bits. But how many do we need, exactly?
Key Size

- Again, we have a simple formula:
  \[ K = \log_2 N \]

- where \( N \) is the number of key choices and \( K \) is the number of bits needed to represent the key.

- Unfortunately, many people do not know how to perform a base-2 logarithm. So, and I am sorry to say this, from Calculus, we have an equivalent equation:
  \[ K = \frac{\log N}{\log 2} \]

- where the log is now just the base-10 log available on all calculators, and also on the Calculator toll in Windows.

- So, for a key range of 100 trillion, we have:
  \[ K = \frac{\log 100,000,000,000,000}{\log 2} = \frac{14}{0.301} = 46.51 = 47\text{bits} \]

- Yep... 47 is between 32 and 64.
How many Bits is Enough?

- That is the big question. So far, by pretending, we have seen that a 47-bit key can be cracked in around 11 and a half days. One initial encryption standard used a 56-bit key. Based on our pretend assumptions, that key would require over 22 years to crack.
- How long for the 64-bit key, using our same assumptions? I come up with an astonishing 58 centuries! Does that seek safe enough to you? It should be. I would be happy with 58 centuries.
- But others are not, such as governments, large corporations, and terrorists. They want even stronger encryption. 1024 bits. 4096 bits. Staggeringly large key spaces.
- Why? Because the 58 centuries we get for the 64 bit key have actually been reduced to around 5 minutes, as researches and hackers have discovered techniques that analyze special packets within a encrypted stream and mathematically deduce the key. Magic with math.
- So, maybe now 1024 bits or more looks a lot more attractive now.
How many Bits is Enough?

• Here is a list of encryption algorithms and some of their key sizes.
Now, What About the Technique?

• The size of the key is one factor affecting the time required to crack an encryption code. Another factor is the algorithm used in the encryption process. Let’s look at some simple algorithms and see how their performance can be classified.

• Here is one way to implement the Caesar Shift:

```c
for ( p = 0; p < N; p++) {
    charout = alphabet[ charin[p] + shiftval ];
}
```

• where N is the length of the charin data. The important thing here is not the statement that finds the new charout value, but rather the fact that the for-loop makes N passes through the data. We say this code has $O(n)$ execution (Order-n).
Now, What About the Technique?

• To appreciate what $O(n)$ means, let’s look at another algorithm that has an $O(n^2)$ execution time:

```c
for ( x = 0; x < N; x++ )
    for ( y = 0; y < N; y++ )
    {
        statements...
    }
```

• In this nested loop, the inner loop (the y variable loop) makes N passes through the statements for every one pass through the outer loop (the x variable loop). Overall, the statements inside the inner loop will execute $N^2$, or $N^2$ times. So, this type of algorithm is $O(n^2)$. Naturally, we also have algorithms that are $O(n^3)$, $O(n^4)$, etc.
Now, What About the Technique?

- Even more complex algorithms may have $O(2^N)$ execution time. Look at the following table to see why we get worried when the algorithm is $O(2^N)$:

<table>
<thead>
<tr>
<th>N</th>
<th>O(n)</th>
<th>$O(n^2)$</th>
<th>$O(n^3)$</th>
<th>$O(2^N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>25</td>
<td>125</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>1024</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>400</td>
<td>8000</td>
<td>1048576</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>2500</td>
<td>125000</td>
<td>$1.125 \times 10^{15}$</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>10000</td>
<td>1000000</td>
<td>$1.268 \times 10^{30}$</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1000000</td>
<td>1000000000</td>
<td>$1.072 \times 10^{301}$</td>
</tr>
</tbody>
</table>
Now, What About the Technique?

• So why do we worry about the complexity of an encryption algorithm?
• Complexity is good if we are thinking about someone trying to crack our code.
• Complexity is bad if we are using the encryption in real time (such as in a VPN tunnel).... We must not disregard the encryption / decryption overhead at each end of the tunnel.
All Right Then, But What is Hashing?

- Encryption and hashing are different things, but both provide a secure method of transforming information. The difference is that data that is encrypted may be decrypted at a later time, using the secret key. But when a message is hashed, it is converted into a different form and can not be turned back (Unhashed? Dehashed?) into the original message.

- As it turns out, this is OK and actually quite useful. First, the time to encrypt a message and the time to hash a message may be drastically different, with the hash typically taking less time. Also, there are times when we are not interested in recovering the original message. For example, when you enter a password on a web page, you can either encrypt the password and compare the result with a stored list of encrypted passwords, or you can hash the password and search a list of hashed passwords. Since you do not need to ever recover the original password, why use encryption?
All Right Then, But What is Hashing?

- As a simple example, suppose our hash algorithm takes an input string and adds the ASCII values of the first letter in the string, the middle letter, and the last letter. Then we save only the lower 6 bits of the sum, giving a range of 0 to 63 for the hash value.

- Here are some examples:

<table>
<thead>
<tr>
<th>String</th>
<th>First Letter And Value</th>
<th>Middle Letter And Value</th>
<th>Last Letter And Value</th>
<th>Sum</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>James</td>
<td>J - 74</td>
<td>m - 109</td>
<td>s - 115</td>
<td>298</td>
<td>42</td>
</tr>
<tr>
<td>Antonakos</td>
<td>A - 65</td>
<td>n - 110</td>
<td>s - 115</td>
<td>290</td>
<td>34</td>
</tr>
<tr>
<td>GREEN</td>
<td>G - 71</td>
<td>E - 69</td>
<td>N - 78</td>
<td>218</td>
<td>26</td>
</tr>
<tr>
<td>CST!104</td>
<td>C - 67</td>
<td>! - 33</td>
<td>4 - 52</td>
<td>152</td>
<td>24</td>
</tr>
<tr>
<td>EAT</td>
<td>E - 69</td>
<td>A - 65</td>
<td>T - 84</td>
<td>218</td>
<td>26</td>
</tr>
</tbody>
</table>
All Right Then, But What is Hashing?

- Whoops! Do you see that two different strings (GREEN and EAT) hash to the same value? This is called a collision. By itself a collision is not a problem, because we can modify the data structure where we store our hash values to keep stack of more than one string per hash value. What is disturbing about collisions is that you can use a different string than is intended to obtain the same hash value. What this means is that if your password is saved as a hash value, it may be possible for someone to use a different sequence of symbols to get into your account, without having to use your exact password.
All Right Then, But What is Hashing?

• This brings up an even more disturbing point. We can not predict in advance when a collision might occur or how often. It all depends on how well you construct your hash algorithm. You will hopefully agree in this case that only using the first, middle, and last symbols of the input string may not lead to a good hash value since we are ignoring other symbols in the string. So, a better hash algorithm would add all the ASCII values of the string symbols.
All Right Then, But What is Hashing?

• An even better hash algorithm will combine the string symbols in different, creative ways and result in even larger hash values, further reducing the number of collisions.

• Two widely used hash algorithms are MD5 (Message Digest 5) and SHA1 (Secure Hash Algorithm 1). MD5 creates a 128 bit hash and SHA1 creates a 160 bit hash. Here are the hash values from each algorithm for a sample data file (actually for the file containing these lecture notes, before adding the screen shot of the hash program and writing any more text:
Hashing Examples

• Short ASCII text file for testing hashing utility. Note the ‘0’ characters at the beginning and end.

Whoops! Do you see that two different strings (GREEN and EAT) hash to the same value? This is called a collision. By itself a collision is not a problem, because we can modify the data structure where we store our hash values to keep stack of more than one string per hash value. What is disturbing about collisions is that you can use a different string than is intended to obtain the same hash value. What this means is that if your password is saved as a hash value, it may be possible for someone to use a different sequence of symbols to get into your account, without having to use your exact password.

This brings up an even more disturbing point. We can not predict in advance when a collision might occur or how often. It all depends on how well you construct your hash algorithm. You will hopefully agree in this case that only using the first, middle, and last symbols of the input string may not lead to a good hash value since we are ignoring other symbols in the string. So, a better hash algorithm would add all the ASCII values of the string symbols.

An even better hash algorithm will combine the string symbols in different, creative ways and result in even larger hash values, further reducing the number of collisions. Two widely used hash algorithms are MD5 (Message Digest 5) and SHA1 (Secure Hash Algorithm 1). MD5 creates a 128 bit hash and SHA1 creates a 160 bit hash. Here are the hash values from each algorithm for a sample data file (actually for the file containing these lecture notes, before adding the screen shot of the hash program and writing any more text):
Hashing Examples

• MD5 and SHA1 hash values for the sample text file:
Hashing Examples

- Now I just change the ‘0’ on the first line to a ‘1’. Note that this is a one bit change!

Whoops! Do you see that two different strings (GREEN and EAT) hash to the same value? This is called a collision. By itself a collision is not a problem, because we can modify the data structure where we store our hash values to keep stack of more than one string per hash value. What is disturbing about collisions is that you can use a different string than is intended to obtain the same hash value. What this means is that if your password is saved as a hash value, it may be possible for someone to use a different sequence of symbols to get into your account, without having to use your exact password.

This brings up an even more disturbing point. We cannot predict in advance when a collision might occur or how often. It all depends on how well you construct your hash algorithm. You will hopefully agree in this case that only using the first, middle, and last symbols of the input string may not lead to a good hash value since we are ignoring other symbols in the string. So, a better hash algorithm would add all the ASCII values of the string symbols.

An even better hash algorithm will combine the string symbols in different, creative ways and result in even larger hash values, further reducing the number of collisions. Two widely used hash algorithms are MD5 (Message Digest 5) and SHA1 (Secure Hash Algorithm 1). MD5 creates a 128 bit hash and SHA1 creates a 160 bit hash. Here are the hash values from each algorithm for a sample data file (actually for the file containing these lecture notes, before adding the screen shot of the hash program and writing any more text):

0
Hashing Examples

So cool! The MD5 and SHA1 hash values are widely different, not just a little different. Perhaps this is one reason why MD5 and SHA1 are accepted algorithms for use verifying the integrity of digital forensic evidence.
Hashing Examples

• Now the ‘1’ on the first line is changed back to a ‘0’ and the ‘0’ on the last line is changed to a ‘1’. Another *one bit* change!

Whoops! Do you see that two different strings (GREEN and EAT) hash to the same value? This is called a collision. By itself a collision is not a problem, because we can modify the data structure where we store our hash values to keep stack of more than one string per hash value. What is disturbing about collisions is that you can use a different string than is intended to obtain the same hash value. What this means is that if your password is saved as a hash value, it may be possible for someone to use a different sequence of symbols to get into your account, without having to use your exact password.

This brings up an even more disturbing point. We cannot predict in advance when a collision might occur or how often. It all depends on how well you construct your hash algorithm. You will hopefully agree in this case that only using the first, middle, and last symbols of the input string may not lead to a good hash value since we are ignoring other symbols in the string. So, a better hash algorithm would add all the ASCII values of the string symbols.

An even better hash algorithm will combine the string symbols in different, creative ways and result in even larger hash values, further reducing the number of collisions. Two widely used hash algorithms are MD5 (Message Digest 5) and SHA1 (Secure Hash Algorithm 1). MD5 creates a 128 bit hash and SHA1 creates a 160 bit hash. Here are the hash values from each algorithm for a sample data file (actually for the file containing these lecture notes, before adding the screen shot of the hash program and writing any more text):
Hashing Examples

- And again we see radically different MD5 and SHA1 hashes... showing that it matters *where* the data is changed inside a file in addition to *how* the data is changed.
Malware and Hashing

- Malware writers use hashing methods to obscure their code and make it harder to reverse engineer.
Malware and Hashing

• The hashing routine creates 32-bit hash codes based on the names of exported DLL functions:
Malware and Hashing

- Sequence of DLL calls performed during execution of malware to download a file from the Internet and execute it.

<table>
<thead>
<tr>
<th>Hash Value</th>
<th>DLL Function</th>
<th>DLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A412FD89</td>
<td>LoadLibraryA</td>
<td>KERNEL32.DLL</td>
</tr>
<tr>
<td>E4EC2161</td>
<td>URLDownloadToCacheFileA</td>
<td>URLMON.DLL</td>
</tr>
<tr>
<td>2D6D019</td>
<td>LocalAlloc</td>
<td>KERNEL32.DLL</td>
</tr>
<tr>
<td>C5FF2F46</td>
<td>VirtualProtect</td>
<td>KERNEL32.DLL</td>
</tr>
<tr>
<td>410E2A69</td>
<td>MoveFileA</td>
<td>KERNEL32.DLL</td>
</tr>
<tr>
<td>16EF74B</td>
<td>WinExec</td>
<td>KERNEL32.DLL</td>
</tr>
<tr>
<td>D6196BE1</td>
<td>RtlExitUserThread</td>
<td>NTDLL.DLL</td>
</tr>
</tbody>
</table>
Thank you!

James L. Antonakos
james@whitehatforensics.com
(607) 765-2686