This is a reformat of https://stichintime.wordpress.com/2009/04/09/rainbow-tables-* Posted by paul under Rainbow Tables, Security | Tags: Rainbow Tables, Security |

# Rainbow Tables: Introduction

In January, I took the SANS Security Essentials (401) class at SANS West. In the class, we briefly covered the concept of rainbow tables. Rainbow tables are (superficially) just password lookup tables: if you know the encrypted password, you can look it up in the table and get the unencrypted password. (The word 'superficially' there is critical – rainbow tables are far more interesting than simple lookup tables.)

One of the evening sessions (after the boot camp session) featured Ed Skoudis talking about penetration testing. He also talked about rainbow tables. What really caught my attention was his discussion of chains to vastly reduce the size of the tables. The chains provide a clever way to trade off table size at the cost of some processing time. I just had to understand the details, and I didn't have the patience to wait until some day in the future when I might take Ed's Pen Testing class. I cornered Ed at the bar on the last night, asked how he would recommend learning them, and he pointed me to Wikipedia.

The wikipedia article is great, but a bit tough to read for someone new to the concept. Fortunately, the article refers to another article, How Rainbow Tables Work, which is a bit easier. It also is a good article, and the two were all I needed to help me build some samples of my own in order to really understand the concepts.

In this series of posts, I will attempt to provide an even easier introduction to rainbow tables. I hope that the posts will help you understand the other articles, and appreciate the beauty of the approach.

I've broken the discussion into six parts, so that you can skip any bits that are already familiar to you. This introduction is Part 1. Part 2 briefly talks about encryption and introduces the concept of hashes. Part 3 looks at some really simple hashing algorithms, including the one used for the examples here. Part 4 goes into reduction functions. Finally, Part 5 explains rainbow tables and chains. If you understand that password encryption uses hashes and are familar with common hashes like MD5 and SHA-1, you can probably jump right to Part 5, and then read the other parts if you want to fill in any gaps.

These posts contain simple examples that you can work through yourself with a spreadsheet. I have built one for you, which you can view using Google Docs or download to Excel. I encourage you to play with the values in the spreadsheet if you really want to get a deep understanding of how each part works.

If you've never used Google Docs before, it has the equivalent of Excel's tabs. At the bottom, you'll see links that will take you to the different pages. You also can select File – Export – .xls to export it into an Excel Spreadsheet, and you'll find all the tabs there.

## Encryption and Hashes

As I mentioned in the first post of this series, rainbow tables are used to find a password if you know the encrypted password.

Passwords typically are (or should be) encrypted with a one-way encryption algorithm. This type of encryption is known as a hash. With a hash, there is no algorithm that can be applied to the encrypted password to determine the unencrypted password. There are only two ways to determine the unencrypted password:

- keep trying passwords until you find the right one (brute force)
- in advance, create a list of passwords and their encrypted results. This is known as a lookup table. Such a table can be huge, but is very simple to use and is fast. Rainbow tables are a compromise. They consume less space, but require more processing. Compared with brute force, they still can be very fast (unless the password is poor and can be guessed immediately).

To understand rainbow tables, you need to be comfortable with hashes. Most experienced computer users are at least somewhat familiar with MD-5 hashes, which are often used as a checksum to validate that a downloaded file was not corrupted in-flight. MD-5 is known to be vulnerable, but it makes a fine checksum. SHA-1 is a more secure hash.

The rainbow table explanations I cited before (wikipedia and kuliukas) both use MD-5 for their example hashes. While MD-5 is well-known and is easily available, it is difficult to calculate an MD-5 hash within Excel. Because Excel makes a great platform for experimenting with data and tables to learn a concept, I found some simpler hash functions that I'll use for the examples here. These hashes would be terrible for real encryption, but they work well for creating a simple, understandable rainbow table in Excel. We'll look at those hash algorithms in Part 3.

If you're not comfortable with hashes, these simple algorithms will also provide a gentle introduction to the concept.

The examples in the other articles use character strings for their passwords to hash. To keep things simple, my examples will stick to encrypting numbers between 0 and 99. After all, in a computer, characters are all represented by numbers anyway, so we're just skipping the step of translating a character into ASCII or Unicode.

Let's go look at these simple hashes in Part 3.

# Rainbow Tables – Part 3 (Simple Hashes and Collisions)

Posted by paul under Rainbow Tables, Security | Tags: Rainbow Tables, Security |
[3] Comments

This is Part 3 of a five part series:

1. Introduction
2. Encryption and Hashes
3. Simple Hashes and Collisions (you are here)
4. Reduction Functions
5. Rainbow Tables and Chains

Geoff Kuenning, a computer science professor at Harvey Mudd College, has a great web page about hashes as a part of one of his classes. Let's look at two of the three simple hash functions he presents (we'll use one of these for our rainbow table).

*If you haven't looked at the spreadsheet yet, now is probably a good time to do so. It is broken into tabbed pages. The first four tabs are dedicated to Kuenning's three hash functions.*

When we look at the hash functions, one of the things we want to consider is the number of *collisions* that occur. Part of the nature of hash functions is that multiple input values can produce the same output result. When this happens, it is known as a collision. (If you remember high-school trigonometry, this is like the sin() and cos() functions. sin(0) and sin(180 degrees) both equal zero.)

*Note:* My nomenclature differs slightly from Kuenning's. I use H() to indicate the hash function, h to indicate the hashed value, and p to indicated the unencrypted password (aka plaintext). On this page only, I'll make p and h orange so you remember that we're starting with one and calculating the other.

## Hash Function 1: The Division Method

The division method simply uses the modulo function:

```
h = p mod m
```

where m is a prime number. (m should be far from a power of 2, but for our purposes it doesn't matter.)

here is a table of hash values for plaintext values from 0-19, with a m value of 13:

| p | h | p | h | p | h | p | h |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 5 | 5 | 10 | 10 | 15 | 2 |
| 1 | 1 | 6 | 6 | 11 | 11 | 16 | 3 |
| 2 | 2 | 7 | 7 | 12 | 12 | 17 | 4 |
| 3 | 3 | 8 | 8 | 13 | 0 | 18 | 5 |
| 4 | 4 | 9 | 9 | 14 | 1 | 19 | 6 |

Lousy for encryption, don't you agree? But, it *is* a hash – you can't tell what the plaintext value is if you only know the result. And, there are collisions.

I'm going to skip the next function that Kuenning presents, the Knuth variant on division. It is easy to understand, and is included in the spreadsheet if you want to see how it compares with the other two.

## Hash Function 2: The Multiplication Method

This hash function is a little more involved, but is still simple enough to implement easily in a spreadsheet. Kuenning conveniently breaks it into three calculations. This makes it easier to understand (and easier to program).

```
s = p*A
x = fractional part of s
h = floor(m*x)
```

Below is a table of the results of this hash applied to the numbers 0-19, with A set to 6.213335, m set to Kuenning's recommended value of (SQRT(5) – 1)/2, and x truncated to 4 digits (if s = 1.234567, x is then 2345)

| p: | s: | x: | Hash: | p: | s: | x: | Hash: |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0000 | 10 | 62.13335 | 1333 | 0823 |
| 1 | 6.213335 | 2133 | 1318 | 11 | 68.346685 | 3466 | 2142 |
| 2 | 12.42667 | 4266 | 2636 | 12 | 74.56002 | 5600 | 3460 |
| 3 | 18.640005 | 6400 | 3955 | 13 | 80.773355 | 7733 | 4779 |
| 4 | 24.85334 | 8533 | 5273 | 14 | 86.98669 | 9866 | 6097 |
| 5 | 31.066675 | 666 | 0411 | 15 | 93.200025 | 2000 | 1236 |
| 6 | 37.28001 | 2800 | 1730 | 16 | 99.41336 | 4133 | 2554 |
| 7 | 43.493345 | 4933 | 3048 | 17 | 105.626695 | 6266 | 3872 |
| 8 | 49.70668 | 7066 | 4367 | 18 | 111.84003 | 8400 | 5191 |
| 9 | 55.920015 | 9200 | 5685 | 19 | 118.053365 | 533 | 0329 |

A few points are worth noting:

- We don't see any collisions here. However, the spreadsheet shows that there are collisions.
- No matter what integer we use as the source, the resulting hash will always be a fixed length: 4 digits. In other words, the hash will always be less than or equal to 9999. (Actually, it will be less than 6180).
- Because the hash is always an integer between 0 and 6180, we know that if we have a set of over 6180 numbers, we are guaranteed to have collisions.
- Zero does not hash particularly well with this algorithm.

This is a great hash for creating a simple rainbow table to learn how they work. Before we do so, however, let's look at the notion of reduction functions in Part 4.

# Rainbow Tables – Part 4 (Reduction Functions)

Posted by paul under Rainbow Tables, Security | Tags: Rainbow Tables, Security |
[3] Comments

This is Part 4 of a five part series:

1. Introduction
2. Encryption and Hashes
3. Simple Hashes and Collisions
4. Reduction Functions (you are here)
5. Rainbow Tables and Chains

*Reduction functions* are at the heart of how rainbow tables work.

To understand reduction functions, lets look at a set of two-digit values that are "encrypted" into four-digit hashes. For example, using the sample hashing algorithm described in Part 3, the number 10 is encrypted into 0823:

```
 p    h
10  0823
```

Now, imagine if we had a special function that could somehow map a 4 digit hash to a 2 digit number. Because hashes are irreversible, the two digit number is not the original password. But, it *is* a two digit number, which means it is a valid password. For example, a really simple reduction function could take the last two digits of a four digit number. That would map the hash 0823 into the plaintext 23:

```
   h    p
0823   23
```

That's all a reduction function is – a function that consistently maps a hashed value into a valid plaintext value. It is worth repeating that the reduction funtion is NOT the inverse of the hash. In the example above, 0823 "reduces" to 23, but 23 does not hash into 0823 (23 hashes into 5603).

It is also important that the results of the reduction function be valid plaintext. This is easy to do in our samples where we're dealing with two and four digit numbers. But let's say you are dealing with a system that allows passwords to be only alphanumerics, no special characters. If your reduction function results in characters like !,@,#, or $, then you'll be saying !@#$ because you'll get invalid plaintext!

In part 3, we looked at the fact that all encryption algorithms can have collisions. This is also true for reduction functions. In fact, the situation tends to be worse for reduction functions, because chances are that your maximum password size is much shorter than the size of your hashed values. In our simple example, we're mapping four digit numbers into two digit numbers. Clearly, we'll frequently collide!

The collisions are a real problem for rainbow tables. One strategy to reduce (but not eliminate) their impact is to use multiple reduction functions. In Part 5, we talk about R1 and R2. In those samples, R1 is our function above – the last two digits. R2 is equally simple – it just takes the first two digits.

Let's go look at rainbow tables in Part 5!

## Rainbow Tables – Part 5 (Chains and Rainbow Tables)

Posted by paul under Rainbow Tables, Security | Tags: Rainbow Tables, Security |
[6] Comments

This is Part 5 of a five part series:

So – here we go!  Let's look at rainbow tables!

Here is a sample lookup table for a very simplistic encryption algorithm (discussed in Part 2) that takes a number from 0 to 99 and hashes it into a 4 digit number:

| p1 | h3 |
|---|---|
| 3 | 3708 |
| 10 | 5850 |
| 25 | 4202 |
| 68 | 5520 |
| 89 | 5109 |

It looks like a plain old lookup table with plaintext in the left column, and the hashed value in the right column. But, that isn't the case.  In fact, with the encryption algorithm we're using for the example, the hashed value for 3 is 3955.  So what are we looking at?

As we discussed in Part 1, a basic lookup table would simply list every possible password (plaintext) and its corresponding encrypted value (hash).  If you know the encrypted value, you just look it up and see what the password is.  The problem with an ordinary lookup table is that it can quickly get huge.  A rainbow table provides a way to make it dramatically smaller.  A rainbow table does this through the use of *chains* and *reduction functions.*

The beauty of a rainbow table is that you don't store the whole table. Instead, you store the left-most column and the right-most column, and you calculate the values in between as needed.  The table we saw above contains those two columns (leftmost and rightmost) from this rainbow table:

| p1 | h1=H(p1) | p2=R1(h1) | h2=H(p2) | p3=R2(h2) | h3=H(p3) |
|---|---|---|---|---|---|
| 3 | 3955 | 55 | 4532 | 45 | 3708 |
| 10 | 0823 | 23 | 5603 | 56 | 5850 |
| 25 | 2059 | 59 | 3626 | 36 | 4202 |
| 68 | 3131 | 31 | 3790 | 37 | 5520 |
| 91 | 2554 | 54 | 3213 | 32 | 5109 |

Because we've used 10 entries to represent the 30 entries in table above, we've reduced the space by 66%!  Of course, there is a cost – we have more calculations to do. It is a tradeoff between size and speed.  We'll look into that later; for now, lets just see how the rainbow table works by considering some examples.

Let's say you have an encrypted value of 5520 that you want to decrypt.  Let's look it up in our table. Remember, we've only stored the lookup table, not the full rainbow table, so that is where we need to look:

| p1 | h3 |
|---|---|
| 3 | 3708 |
| 10 | 5850 |
| 25 | 4202 |
| 68 | 5520 |
| 89 | 5109 |

Yay! We have a match! If this were a simple lookup table, we would be done. But it isn't! This lookup table represents the rainbow table above. Because we've saved space, we have some calculations to do. Time to pay the piper… (This is where it gets interesting.)

Right now, all we know is that we have a hashed value of 5520, which is the right-most value in the chain that begins with 68:

| p1 | h1=H(p1) | p2=R1(h1) | h2=H(p2) | p3=R2(h2) | h3=H(p3) |
|---|---|---|---|---|---|
| 68 | 3131 | 31 | 3790 | 37 | 5520 |

So, we take 68 and hash it to get 3131. With hashes, we can always hash a plaintext value, to get its resulting hashed value, but we cannot go backwards (see Part 3). We can't take a hashed value and get its plaintext value. (That's what rainbow tables are for!) Great – so we have 3131. Now what do we do?

Here we meet the really clever part of rainbow tables. Rainbow tables use something called reduction functions to "reduce" the hashed value (4 digits in our case) to a valid plaintext value (2 digits for us). Reduction functions are really important, and are discussed in more detail in Part 4. For now, let's just learn enough about them to be able to walk through our examples. A few important points:

- A reduction function is NOT an inverse of a hash function. Hash functions don't have an inverse.
- But, a reduction function DOES consistently map a hashed value to *some* plaintext value. But the plaintext value is meaningless. (Meaningless, but helpful for the rainbow table!)
- For any given valid hash value (eg, 3131), the reduction function MUST generate a valid plaintext value.
- Just like hashes have collisions, so do reduction functions. Because of this, rainbow tables use multiple reduction functions. (More about this in Part 4.)

So, for our example, we apply our reduction function, R1, to the value we got from encrypting 68. R1(3131) = 31. Next, we hash 31: H(31)=3790. We pause, check if that is the value we started with (5520), notice that it is not, and repeat the steps.

That is, we apply our second reduction function, R2, to 3790. R2(3790) = 37. We hash 37, which results in H(37)=5520. We pause, check if this is the value we started with (5520), and it is! So we stop, knowing that 37 hashes into 5520. So our plaintext is 37.

Boy, that was a lot of work, especially given that we had a match in our table! Yes, that's true. Remember – we're doing extra work to save space.

So let's recap the algorithm we have seen so far:

```
1. Find the hashed value in the lookup table.
2. Take the plaintext value and hash it.
3. Does that hash match the hash we have?
   If so, stop. The value you just hashed is the value you're looking for.
4. If not, apply the reduction function to get a new plaintext value,
   and go back to step 2.
```

So what happens if the hashed value we have isn't in the lookup table? For example, what would we do if we started with the hashed value of 3626? Step 1 in our algorithm obviously isn't sufficient!

The solution is to apply the reduction function. Since this is essentially walking backwards through the chains, we apply the last reduction function (R2). What is R2(3626)? Well, you don't know (unless you've read Part 4), but that is okay. Let R1 and R2 be "black boxes" for now and take my word for it – R2 (3626) is 36. (Take a close look at what you've seen so far for R1 and R2 – you might be able to guess the algorithms.) Hash that number, H(36)=4202, and try the algorithm again. Looking back at the lookup table (not the full rainbow table), this time we find 4202. We see that its corresponding value for p1 is 25. Now we can go on to step 2: H(25)=2059. Step3: is 2059 the number we're looking for? No, we looking for 3626, so on to step 4:

R1(2059)=59. Back to step 2: H(59)=3626. Step 3: s 3626 the number we're looking for? Yes! Therefore, 59 is its plaintext.

So, let's rewrite the algorithm a little bit:

```
1. Find the hashed value in the lookup table.  If you find it, go to step 2.
  If not:
  1a. Starting with the last reduction function (e.g., R2), "reduce" the
      hashed value to get a new plaintext number. Every time you repeat
      step 1, you go to the next lowest reduction function (e.g., R2,
      then R1).
  1b. Hash the new plaintext number and repeat step 1 from he beginning
      with this new hash value.
2. Take the plaintext value and hash it.
3. Does that hash match the hash we have?
   If so, stop. The value you just hashed is the value you're looking for.
4. If not, apply the reduction function to get a new plaintext value, and
   go back to step 2.
```

Essentially, step 1 backs you up column-by-column in the rainbow table until you find a hash and can match a row. Then, steps 2-4 move you forward through a specific row to obtain the value you need. (And if you don't find a value in these steps, then your rainbow table doesn't have the information you're looking for.)

Try it on your own, perhaps using 2554.

So, there you go – a gentle introduction to Rainbow Tables. Hopefully this will help make other descriptions (such as those at wikipedia and kuliukas) a bit easier. I encourage playing around with the spreadsheet and walking through the process with other sample values.

## Addendum

I'll update this more later (or just replace it), but wanted to comment on the fact that there is an important difference between my sample rainbow tables and those in the Wikipedia article. in my rainbow tables, the leftmost column is plaintext, and the rightmost is a hash. In the wikipedia article, the rightmost column is also plaintext. It is the same table, but they do one extra reduction (R3) to get the plaintext. This means that you can't look up your hashed value in the table – the first thing you MUST do is run your last reduction, and then look up the resultant plaintext. Why would you do this? Space. In my example (and the wikipedia example), the plaintext and hashes are similar sizes (and small). In reality, if you're dealing with MD5 or SHA-1 hashes, your plaintext values are going to be MUCH shorter! So, you can save a lot of space by doing one more reduction, and using the reduced values instead. Of course, you increase the risk of collisions (and run extra calculations), but that is the price of admission.