

Rainbow tables

Discover how to crack password hashes using a colourful collection of chains

In this issue...

▶ WHAT'S COVERED

The best way of ensuring the security of stored passwords is not to store them, but instead to store a cryptographic hash of the password. From the hash it's extremely hard to guess the password. In general, finding out what the password was using just the hash requires a brute-force attack of trying every possible password. Rainbow tables are an efficient method of using gigabytes of pre-computed data in order to calculate a password given its hash.

Stronger passwords

If you have no influence on how your passwords are hashed and stored in a system, what can you do to make your password hashes immune from rainbow table attacks? The first method is obvious: use passphrases and not passwords. In other words, 'whatlightthroughyonderwindow breaks' is a more secure password than 'romeo'. Another (more sneaky) way is to pepper your password with characters that don't appear on your keyboard. Learn the [ALT]+numpad sequences, like [ALT]+0222 for the Þ (thorn) character, or [ALT]+0169 for the copyright symbol ©. This assumes that the system will accept such characters, but most systems should have moved on from accepting only alphanumeric letters. ■

Storing passwords in a database is an extraordinarily bad thing to do. Once the bad guys get access to the database, it's all over. The security of your systems – and your customers – is severely compromised. This is why, in the event that you forget a password to an online service, the best websites generate a temporary password and send that to you instead: they simply aren't storing the original. But systems still need to authenticate users via their password. To get round this problem, most systems take the password that the user enters, calculate a cryptographic hash of it (the usual algorithms are MD5 and SHA-1), and then check the hash against the one that's stored in the database.

You might think that's just moving the problem somewhere else, but that's not the case: one of the great benefits of cryptographic hashes is that given a hash, calculating what the original password was is extremely difficult. (The other benefit of cryptographic hashes is that finding two strings that hash to the same value is also extremely difficult.)

Until quite recently, the only way to crack a given hash was using a brute-force attack: generating all the passwords possible, hashing each one and comparing the hashes to the one being cracked. This method is very unpractical. Using an alphabet of 62 characters – the upper and lower case letters and digits – to crack a password of seven characters, there are 3,521,614,606,208 possible combinations. Even if we could test a million passwords a second, it would take 41 days to try them all.

That was the case until six years ago, when Philippe Oechslin published *Making a Faster Cryptanalytic Time-Memory Trade-Off* at the Advances in Cryptology CRYPTO 2003 conference. This paper was the introduction of 'rainbow tables', which has become the method for cracking password hashes.

The biggest issue with the standard brute-force attack is that the algorithm has no memory. It's a long drudge of permuting the next possible password, calculating its hash and checking whether that is equal to the hash being cracked, over and over again.

In contrast, the rainbow table algorithm pre-computes a very large table of passwords and hashes, then uses that in the cracking process. It's still a brute-force process, but it's now guided by this pre-computed data, which reduces the processing time considerably. It's an example of the principle that to achieve more speed, you use more memory.

Reduction function

The algorithm builds on a previous result devised by Martin Hellman in 1980. Hellman's algorithm makes use of an inverse function of the hash function known as the 'reduce function'. We've already discussed that there is no 'real' inverse function (that is, a function that given a hash calculates the password), but the reduce function is not that. It's just a function that takes a hash value and generates a password out of it. By no means is this the original password, and indeed the reduce function doesn't even attempt to calculate it. ▶

“ The rainbow table algorithm pre-computes a large table of passwords and hashes to use in the cracking process ”

Reduce functions

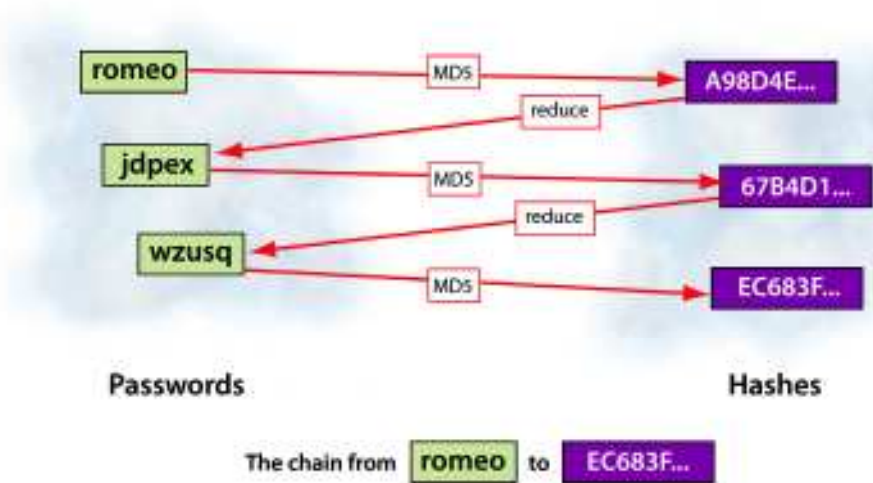
A rainbow table is dependent on thousands of reduce functions. So how is a reduce function devised? One simple class of reduce functions takes successive bits from the hash and converts them to characters. For example, if we assume that our password uses seven characters from a 62-character alphabet, the first reduce function would peel off bits from the hash, six at a time, to form the characters of the password (and would use 42 bits in all). The second reduce function would then drop the first bit before peeling off bits like the first. And so on. We could also take every second bit or every third bit (and so on) for another set of reduce functions. ■

- ▶ Let's use an example. Suppose that we're trying to crack the hash made from a password made up of seven capital letters. An example reduce function (and an absolutely terrible one, by the way) would be to take the string value of the hash in hex and create the password that consists of the first seven letters in the hex string. (It's not a very good reduce function since there would only be six possible letters – A through F – to make the password.)

We start off with a random password from our universe of passwords. We hash it to produce its hash value and then reduce this hash value to produce a new password. We hash this new password to create another hash. And then we reduce this new hash to form another password. We repeat these steps for a certain number of cycles (say 1,000 times) to reach a final hash value. Finally we throw away all the intermediate steps to leave the initial random password and the chain's final hash.

The important part of Hellman's process is the chain. The initial password and final hash (along with the specific reduce function) define a whole chain of passwords and hashes without having to store the intermediate values. If we need an intermediary value, we start at the beginning and hash/reduce until we get to it.

If we pre-compute a million such chains, we will in effect have 1,000 million passwords and hashes only taking up the space occupied



▲ The hash-reduction chain is key to the success of hash-cracking algorithms.

by a million start/end pairs. If we assume ASCII values for letters and the 16-byte MD5 hash, we'd essentially need about 24MB of memory to store those start/end pairs (8MB for the passwords, 16MB for the hashes).

It's not that much really, so let's go for broke. We'll pre-compute 100 million chains for this reduce function. We'll end up with about 800MB for the initial passwords, and 1.6GB for the final hashes. We will have covered 100 billion passwords and hashes.

To crack a hash using this system, first compare the given hash against all of the final hashes. If it's there, the password is going to be the last one in the chain – so all you need to do is recalculate the chain to find it.

If the hash isn't there, reduce it and then hash the result. Now check to see if this calculated hash is in the list of final hashes. If it's there, the password is going to be the second-to-last password in the chain, and a brief recalculation of the chain will reveal it.

This system sounds perfect. A 2.4GB pre-calculated database of 100 billion passwords, and all we need to do to find the password is calculate a maximum of 1,000 hashes at run-time. It sounds very efficient.

Unfortunately, there is a fly in the ointment – and it's large enough to make this method pretty much non-viable in practice.

Merged chains

When we were describing the chains, we assumed that we were using a perfect reduce function. Put simply, we assumed that given a hash value our reduce function would produce a unique password. But the only perfect reduce function is the inverse hash function, and in general the reduce function (through no fault of its own) will return the same password for a wide variety of different hashes. For instance, there are 3.5×10^{12} seven-character alphanumeric passwords and 3.4×10^{38} MD5 hashes. So, approximately, every possible password would be formed by reducing 10^{26} hashes. If that occurs in our 100 million chains, we have a problem: two or more chains will merge and start producing the same passwords and hashes (see Figure 1).

Even worse, the hash/reduce chain starting at the password that we're trying to find may merge with one of the pre-computed chains. This means we may 'see' the hash in our database during the cracking process, leading us to believe we've succeeded, while in reality we're following a different chain that merges to the one in our database.

The greater the number of chains in the table, the higher the probability that there will be one or more merged chains. The longer the chains, the higher the probability that there will be one or more merges.

Since the problem is created by having a large number of chains in the table, one simple fix is to create multiple smaller tables, each with a different reduce function. This works pretty well in practice, but the memory usage grows quite quickly. Another fix is to try and identify and then remove merged chains during the table generation.

Multiple reduce functions

During the 20 years following Hellman's paper, these and other fixes were devised in order to detect and alleviate merging.

Spotlight on... LAN Manager hashes

The holy grail of hash cracking is the LAN Manager hash (LM hash). This is the method used by Microsoft LAN Manager as well as various Windows OS implementations (up to but not including Windows Vista) to hash and store all user passwords that are shorter than 15 characters long.

The LAN Manager hash is a weak algorithm that uses strong encryption. Passwords are uppercased, padded to 14 characters and split into two halves. Each half is then used to produce a DES key for encrypting a constant string. Next, the two encrypted parts are concatenated to produce the hash. This is a weak algorithm because of the uppercasing (which reduces the 'alphabet' available for the

passwords) and the splitting operation (the hash can be cracked in two easier parts).

Prior to 2003, it took a few hours to crack LAN Manager hashes by brute force, but in that year, Oechslin released a program that used rainbow tables to guide the cracking. The program, called Ophcrack, is able to exploit the weaknesses of the LM hash algorithm and crack 99.9 per cent of hashes in a few seconds. The rainbow table size is a mere 388MB if the program cracks the two halves of a hash separately, or 733MB if the program cracks the entire hash in one go. In the latter case, Ophcrack is about four times faster, so if you have the memory to hold the entire table at once, this is the option to take. ■

“ Rainbow tables are more effective than Hellman’s system because there is a smaller probability of merges occurring ”

But despite these fixes, merging problems remained. This limited the effectiveness of Hellman’s system.

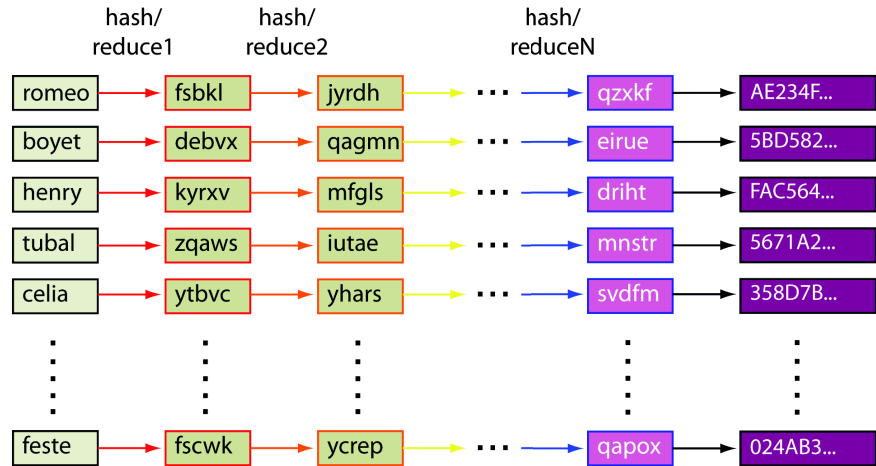
Then along came Oechslin. His method, which is very simple in concept, drastically shrank the probability that any chains in the table would merge.

The basic idea was this: instead of having just one reduce function, have many – but not to generate a table for each reduce function as in Hellman’s sense. Instead, Oechslin’s idea was to make every link in the chain a separate reduce function. Each chain would then use the same reduce functions in the same order.

Figure 2 shows this concept at work. On the left side there’s a set of passwords, one for each chain. First, we hash and reduce using the first reduce function to produce the next password in the chain. We’ll colour this first link red. Next, we hash and reduce using the second reduce function to produce the next password in the chains. We’ll colour this second link orange. We continue like this until we’ve used all of the reduce functions (with the last link being the colour violet).

So why does using the multiple reduce functions and colouring the results like the rainbow mean that this table works better than Hellman’s system?

Firstly, there is a smaller probability of merges occurring. For merging to be an issue with a rainbow table, it has to occur in the same coloured column. For example, suppose that two chains produce the same password when they’re reduced. If these matching passwords are in different columns, the very next step will break the merge because



▲ **Figure 2: The passwords are hashed and reduced to produce the next password in the chain.**

they will use a different reduce function will be used on them. In Hellman’s table, no matter where the merge occurred it would never be broken, and the two chains would be the same from then on.

This simple enhancement improves things dramatically. First of all, the amount of space is reduced considerably: if there are n reduce functions then Hellman’s method would require n tables to be constructed. Only one rainbow needs to be created, and so it can be much larger than one of Hellman’s tables while using the same amount of memory.

To crack a hash using a rainbow table, you proceed pretty much you would when using Hellman’s method. First, check to see if the given hash is in the list of hashes. If it is, you know that the password is the last one

in that chain. If the hash isn’t in the list, reduce it with the last reduce function (the violet one), hash it and then check to see if it’s in the list again. If it’s not, reduce the hash with the second-to-last reduce function (indigo, or blue-violet) and hash the result. Perform the check again, and keep going until either the hash is found or you’ve run out of chain to check.

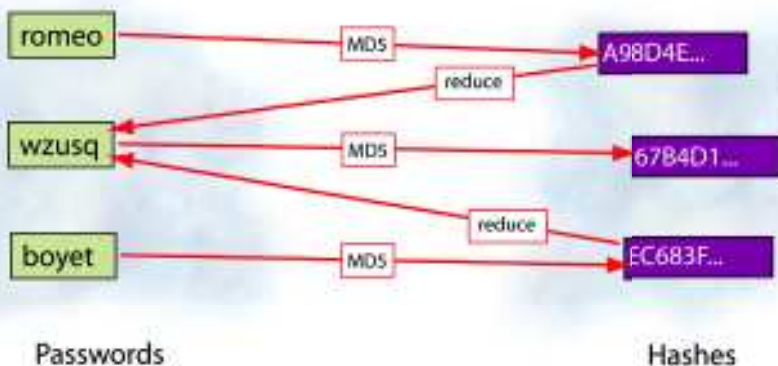
A salty solution

Now that we’ve seen how efficient rainbow tables can be, what can we do to circumvent them in our systems? How can we make our stored hashes secure again?

The answer to this particular question turns out to be very easy. Instead of simply hashing the password when storing it in the database, we salt it by concatenating another string onto the end of the password and then hashing it with MD5. For example, instead of hashing the original password ‘romeo’, we hash ‘romeotakethatyourainbowtable!’.

If the salt is long enough (and random enough and includes enough non-keyboard characters), the ‘password plus salt’ combination will mean that it’s not viable to use the rainbow table to crack passwords. Since rainbow tables always assume that there is no salting going on, using them to crack a salted security system is doomed to failure, unless the salt can be discovered or its generator acquired. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express. feedback@pcplus.co.uk



The chains from romeo and boyet merging

▲ **Figure 1: If two hashes reduce to the same word, the chains will merge and the password could be lost.**