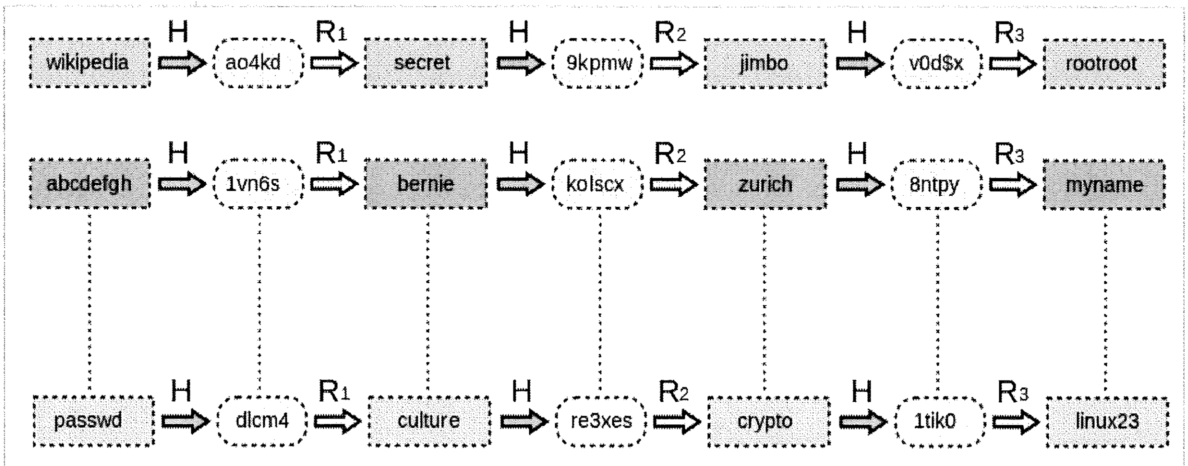
Rainbow table

From Wikipedia, the free encyclopedia

A **rainbow table** is a precomputed table for reversing cryptographic hash functions, usually for cracking password hashes. Tables are usually used in recovering a plaintext password up to a certain length consisting of a limited set of characters. It is a practical example of a space/time trade-off, using less computer processing time and more storage than a brute-force attack which calculates a hash on every attempt, but more processing time and less storage than a simple lookup table with one entry per hash. Use of a key derivation function that employs a salt makes this attack infeasible.

Rainbow tables are an application of an earlier, simpler algorithm by Martin Hellman.[1]



Simplified rainbow table with 3 reduction functions

Contents

Background

Any computer system that requires password authentication must contain a database of passwords, either hashed or in plaintext, and various methods of password storage exist. Because the tables are vulnerable to theft, storing the plaintext password is dangerous. Most databases therefore store a cryptographic hash of a user's password in the database. In such a system, no one—including the authentication system—can determine what a user's password is simply by looking at the value stored in the database. Instead, when a user enters his or her password for authentication, it is hashed and that output is compared to the stored entry for that user (which was hashed before being stored). If the two hashes match, access is granted.

Someone who gains access to the (hashed) password table cannot merely enter the user's (hashed) database entry to gain access (using the hash as a password would of course fail since the authentication system would hash that a second time, producing a result which does not match the stored value, which was hashed only once). In order to learn a user's password, a password which produces the same hashed value must be found.

Rainbow tables are one tool that has been developed in an effort to derive a password by looking only at a hashed value.

Rainbow tables are not always needed, for there are simpler methods of hash reversal available. Brute-force attacks and dictionary attacks are the simplest methods available, however these are not adequate for systems that use large passwords, because of the difficulty of storing all the options available and searching through such a large database to perform a reverse-lookup of a hash.

To address this issue of scale, reverse lookup tables were generated that stored only a smaller selection of hashes that when reversed could generate long chains of passwords. Although the reverse lookup of a hash in a chained table takes more computational time, the lookup table itself can be much smaller, so hashes of longer passwords can be stored. Rainbow tables are a refinement of this chaining technique and provide a solution to a problem called chain collisions.

Precomputed hash chains

Note: The hash chains described in this article are a different kind of chain from those described in the hash chains article.

Suppose we have a password hash function H and a finite set of passwords P. The goal is to precompute a data structure that, given any output *h* of the hash function, can either locate an element *p* in P such that H(*p*) = *h*, or determine that there is no such *p* in P. The simplest way to do this is compute H(*p*) for all *p* in P, but then storing the table requires Θ(|P|*n*) bits of space, where *n* is the size of an output of H, which is prohibitive for large |P|.

Hash chains are a technique for decreasing this space requirement. The idea is to define a *reduction function* R that maps hash values back into values in P. Note, however, that the reduction function is not actually an inverse of the hash function. By alternating the hash function with the reduction function, *chains* of alternating passwords and hash values are formed. For example, if P were the set of lowercase alphabetic 6-character passwords, and hash values were 32 bits long, a chain might look like this:

$$\mathbf{aaaaaa} \xrightarrow[H]{} \mathrm{281DAF40} \xrightarrow[R]{} \mathrm{sgfnyd} \xrightarrow[H]{} \mathrm{920ECF10} \xrightarrow[R]{} \mathbf{kiebgt}$$

An example for a *reduction function*: Given a 32 bit hash, get the last 6 characters in the hash (it's not related to chain example above).

The only requirement for the reduction function is to be able to return a "plain text" value in a specific size.

To generate the table, we choose a random set of *initial passwords* from P, compute chains of some fixed length *k* for each one, and store *only* the first and last password in each chain. The first password is called the *starting point* and the last one is called the *endpoint*. In the example chain above, "aaaaaa" would be the starting point and "kiebgt" would be the endpoint, and none of the other passwords (or the hash values) would be stored.

Now, given a hash value *h* that we want to invert (find the corresponding password for), compute a chain starting with *h* by applying R, then H, then R, and so on. If at any point we observe a value matching one of the endpoints in the table, we get the corresponding starting point and use it to recreate the chain. There's a good chance that this chain will contain the value *h*, and if so, the immediately preceding value in the chain is the password *p* that we seek.

For example, if we're given the hash 920ECF10, we would compute its chain by first applying R:

$$\mathrm{920ECF10} \xrightarrow[R]{} \mathbf{kiebgt}$$

Since "kiebgt" is one of the endpoints in our table, we then take the corresponding starting password "aaaaaa" and follow its chain until 920ECF10 is reached:

$$\mathbf{aaaaaa} \xrightarrow[H]{} \mathrm{281DAF40} \xrightarrow[R]{} \mathrm{sgfnyd} \xrightarrow[H]{} \mathrm{920ECF10}$$

Thus, the password is "sgfnyd".

Note however that this chain does not *always* contain the hash value *h*; it may so happen that the chain starting at *h* merges with the chain starting at the starting point at some point *after h*. For example, we may be given a hash value FB107E70, and when we follow its chain, we get kiebgt:

$$\mathrm{FB107E70} \xrightarrow[R]{} \mathrm{bvtdll} \xrightarrow[H]{} \mathrm{0EE80890} \xrightarrow[R]{} \mathbf{kiebgt}$$

But FB107E70 is not in the chain starting at "aaaaaa". This is called a *false alarm*. In this case, we ignore the match and continue to extend the chain of *h* looking for another match. If the chain of *h* gets extended to length *k* with no good matches, then the password was never produced in any of the chains.

The table content does not depend on the hash value to be inverted. It is created once and then repeatedly used for the lookups unmodified. Increasing the length of the chain decreases the size of the table. It also increases the time required to perform lookups, and this is the time-memory trade-off of the rainbow table. In a simple case of one-item chains, the lookup is very fast, but the table is very big. Once chains get longer, the lookup slows down, but the table size goes down.

Simple hash chains have several flaws. Most serious if at any point two chains *collide* (produce the same value), they will merge and consequently the table will not cover as many passwords despite having paid the same computational cost to generate. Because previous chains are not stored in their entirety, this is impossible to detect efficiently. For example, if the third value in chain 3 matches the second value in chain 7, the two chains will cover almost the same sequence of values, but their final values will not be the same. The hash function H is unlikely to produce collisions as it is usually considered an important security feature not to do so, but the reduction function R, because of its need to correctly cover the likely plaintexts, can not be collision resistant.

Other difficulties result from the importance of choosing the correct function for R. Picking R to be the identity is little better than a brute force approach. Only when the attacker has a good idea of what the likely plaintexts will be he or she can choose a function R that makes sure time and space are only used for likely plaintexts, not the entire space of possible passwords. In effect R shepherds the results of prior hash calculations back to likely plaintexts but this benefit comes with drawback that R likely won't produce every possible plaintext in the class the attacker wishes to check denying certainty to the attacker that no passwords came from his chosen class. Also it can be difficult to design the function R to match the expected distribution of plaintexts.

Rainbow tables

Rainbow tables effectively solve the problem of collisions with ordinary hash chains by replacing the single reduction function R with a sequence of related reduction functions R_1 through R_k. In this way, for two chains to collide and merge they must hit the same value *on the same iteration*. Consequently, the final values in each chain will be identical. A final postprocessing pass can sort the chains in the table and remove any "duplicate" chains that have the same final value as other chains. New chains are then generated to fill out the table. These chains are not *collision-free* (they may overlap briefly) but they will not merge, drastically reducing the overall number of collisions.

Using sequences of reduction functions changes how lookup is done: because the hash value of interest may be found at any location in the chain, it's necessary to generate *k* different chains. The first chain assumes the hash value is in the last hash position and just applies R_k; the next chain assumes the hash value is in the second-to-last hash position and applies R_{k-1}, then H, then R_k; and so on until the last chain, which applies all the reduction functions, alternating with H. This creates a new way of producing a false alarm: if we "guess" the position of the hash value wrong, we may needlessly evaluate a chain.

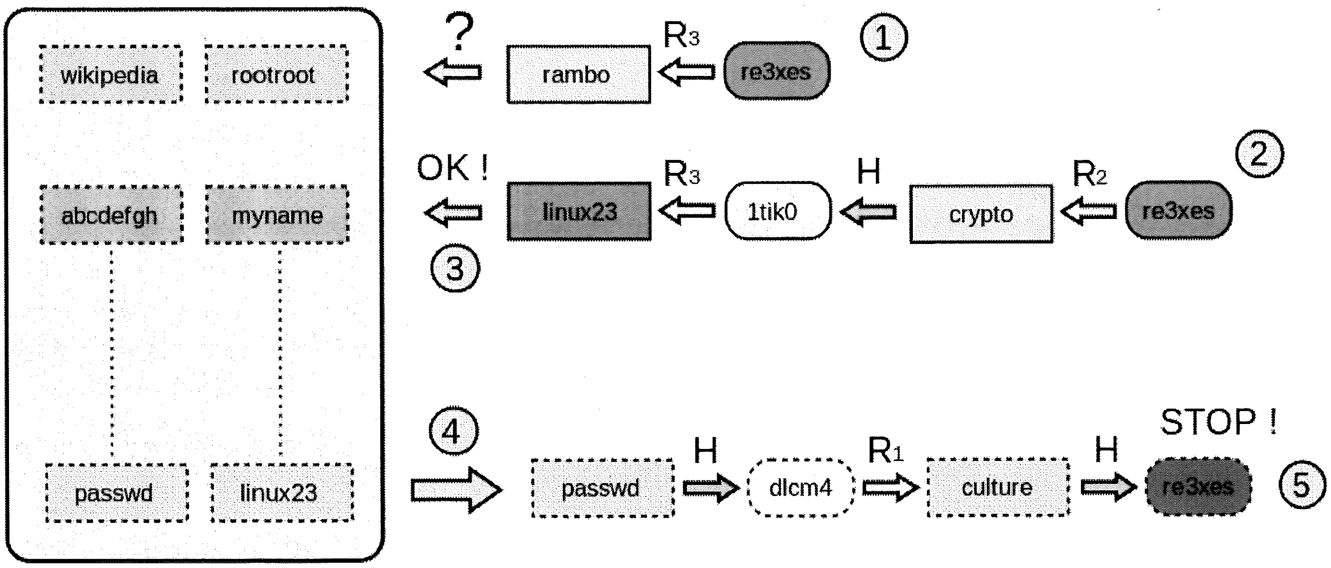
Although rainbow tables have to follow more chains, they make up for this by having fewer tables: simple hash chain tables cannot grow beyond a certain size without rapidly becoming inefficient due to merging chains; to deal with this, they maintain multiple tables, and each lookup must search through each table. Rainbow tables can achieve similar performance with tables that are *k* times larger, allowing them to perform a factor of *k* fewer lookups.

Example

1. Starting from the hash ("re3xes") in the image below, one computes the last reduction used in the table and checks whether the password appears in the last column of the table (step 1).
2. If the test fails (*rambo* doesn't appear in the table), one computes a chain with the two last reductions (these two reductions are represented at step 2)

   Note: If this new test fails again, one continues with 3 reductions, 4 reductions, etc. until the password is found. If no chain contains the password, then the attack has failed.

3. If this test is positive (step 3, *linux23* appears at the end of the chain and in the table), the password is retrieved at the beginning of the chain that produces *linux23*. Here we find *passwd* at the beginning of the corresponding chain stored in the table.
4. At this point (step 4), one generates a chain and compares at each iteration the hash with the target hash. The test is valid and we find the hash *re3xes* in the chain. The current password (*culture*) is the one that produced the whole chain: the attack is successful.



Rainbow tables use a refined algorithm with a different reduction function for each "link" in a chain, so that when there is a hash collision in two or more chains, the chains will not merge as long as the collision doesn't occur at the same position in each chain. As well as increasing the probability of a correct crack for a given table size, this use of multiple reduction functions approximately doubles the speed of lookups.[2]

Rainbow tables are specific to the hash function they were created for e.g., MD5 tables can crack only MD5 hashes. The theory of this technique was first pioneered by Philippe Oechslin[3] as a fast form of time/memory tradeoff,[2] which he implemented in the Windows password cracker Ophcrack. The more powerful RainbowCrack program was later developed that can generate and use rainbow tables for a variety of character sets and hashing algorithms, including LM hash, MD5, SHA1, etc..

In the simple case where the reduction function and the hash function have no collision, given a complete rainbow table (one that makes you sure to find the corresponding password given any hash) the size of the password set $|P|$, the time T that had been needed to compute the table, the length of the table L and the average time t needed to find a password matching a given hash are directly related:

$$T = |P|$$

$$t = \frac{|P|}{2L}$$

Thus the 8-character alphanumeric passwords case ($|P| \simeq 3.10^{12}$) would be easily tractable with a personal computer while the 16-character alphanumeric passwords case ($|P| \simeq 10^{25}$) would be completely intractable.

Defense against rainbow tables

A rainbow table is ineffective against one-way hashes that include large salts. For example, consider a password hash that is generated using the following function (where "+" is the concatenation operator):

```
saltedhash(password) = hash(password + salt)
```

Or

```
saltedhash(password) = hash(hash(password) + salt)
```

The salt value is not secret and may be generated at random and stored with the password hash. A large salt value prevents precomputation attacks, including rainbow tables, by ensuring that each user's password is hashed uniquely. This means that two users with the same password will have different password hashes (assuming different salts are used). In order to succeed, an attacker needs to precompute tables for each possible salt value. The salt must be large enough, otherwise an attacker can make a table for each salt value. For older Unix passwords which used a 12-bit salt this would require 4096 tables, a significant increase in cost for the attacker, but not impractical with terabyte hard drives. The SHA2-crypt and bcrypt methods—used in Linux, BSD Unixes, and Solaris—have salts of 128 bits.[4] These larger salt values make precomputation attacks for almost any length of password infeasible against these systems for the foreseeable future.

Another technique that helps prevent precomputation attacks is key stretching. When stretching is used, the salt, password, and a number of intermediate hash values are run through the underlying hash function multiple times to increase the computation time required to hash each password.[5] For instance, MD5-Crypt uses a 1000 iteration loop that repeatedly feeds the salt, password, and current intermediate hash value back into the underlying MD5 hash function.[4] The user's password hash is the concatenation of the salt value (which is not secret) and the final hash. The extra time is not noticeable to users because they have to wait only a fraction of a second each time they log in. On the other hand, stretching reduces the effectiveness of a brute-force attacks in proportion to the number of iterations because it reduces the number of computations an attacker can perform in a given time frame. This principle is applied in MD5-Crypt and in bcrypt.[6] It also greatly increases the time needed to build a precomputed table, but in the absence of salt, this needs only be done once.

An alternative approach, called **key strengthening**, extends the key with a random salt, but then (unlike in key stretching) securely deletes the salt. This forces both the attacker and legitimate users to perform a brute-force search for the salt value.[7] Although the paper that introduced key stretching[8] referred to this earlier technique and intentionally chose a different name, the term "key strengthening" is now often (arguably incorrectly) used to refer to key stretching.

Rainbow tables and other precomputation attacks do not work against passwords that contain symbols outside the range presupposed, or that are longer than those precomputed by the attacker. However tables can be generated that take into account common ways in which users attempt to choose more secure passwords, such as adding a number or special character. Because of the sizable investment in computing processing, rainbow tables beyond fourteen places in length are not yet common. So, choosing a password that is longer than fourteen characters may force an attacker to resort to brute-force methods.

Certain intensive efforts focused on LM hash, an older hash algorithm used by Microsoft, are publicly available. LM hash is particularly vulnerable because passwords longer than 7 characters are broken into two sections, each of which is hashed separately. Choosing a password that is fifteen characters or longer guarantees that an LM hash will not be generated.[9]

Common uses

Nearly all distributions and variations of Unix, Linux, and BSD use hashes with salts, though many applications use just a hash (typically MD5) with no salt. The Microsoft Windows NT/2000 family uses the LAN Manager and NT LAN Manager hashing method and is also unsalted, which makes it one of the most popularly generated tables.