Password Cracking Power – How Passwords Are Stored and Cracked

Author	josh
Date	2023-12-09

Overview

Passwords, for better or worse, serve as the gateway for access to many of our digital accounts. In order to gain access to a website, computer, or other resource you often have to provide a password. Ideally, a password is strong, meaning that it's hard for an attacker to guess that password and thus break into that resource. But password *cracking* is often a key part of gaining access to accounts, whether maliciously or for legitimate penetration testing. Why does one have to "crack" a password in the first place, and how is it done? Let's learn about the password cracking process!

Why Password Cracking?

Why is password cracking a thing in the first place? If an attacker can breach a password database from some service, then don't they have direct access to the passwords themselves? Hopefully not, as passwords should never be stored in *plain text*, meaning stored without first being transformed through a *cryptographic hash function*. Some very, very insecure websites do store passwords in plain text. In that case, once an attacker has access to the database they have access to the passwords directly.

That said, *most* decently configured applications store password *hashes*, not raw passwords. A *hash function* is a one-way function that transforms some *input* (the password) into some *output*. The function gives the same output for a given input, every time. This article isn't a deep dive into hashes and assumes some understanding of them – but let's discuss how they are used for password authentication.

The database of the application stores only the password *hash*, not the password itself. When a user tries to log in with their password, the application hashes the given password and compares it to the hash in the database. If it matches, the user has supplied the correct phrase and is granted access. But if the password is off, even by one bit, the hash function will have a radically different output and won't match. Passwords are often combined with a random *salt* before they are run through the hash function, and the salt is known and stored alongside the password hash.

So, when a password database is breached, the attacker should only have access to password *hashes* and potentially *salts*, but not the raw passwords. How can we work backward from a password hash to the password? The answer is, we can't! A hash function is one way and cannot be reversed, unlike encryption. It turns out, the only way to figure out what password belongs to a particular hash is to guess…a bunch. In order to find the input that gives a specific output for a cryptographic hash, a computer has to try a bunch of possible inputs until the output is matched.

Let’s look at some examples. A common, general purpose cryptographic hash is the *SHA-256* algorithm. It’s used in lots of cryptography. Let’s look at an example password from the *rockyou .txt* breach wordlist:

rockyou →
4980b1f29fa32ff18c95d0ed931fd48e1ad43a729251d6eddb3cece705ed4d05

That said, SHA-256 is pretty fast and thus easier to compute a lot of hashes for. For password storage, it’s better to use a slower, harder to crack algorithm specifically designed for this purpose. The modern *Argon2* algorithm is a good one, that includes built in functionality for a salt.

```
rockyou (with salt foobarbaz) →
   $argon2i$v=19$m=16,t=2,p=1$Zm9vYmFyYmF6$YMoljsb76DqkMZExF1AZZg
```

So, in order to crack this password hash (whether SHA-256 or Argon2) we have to guess a bunch of possible password combinations (or password + salt) until the hash output matches the one we have acquired. Thus enter our various different password cracking techniques.

Password Cracking Process

Brute Force

The most naive, but most exhaustive form of password cracking is what we call “brute forcing”. In a nutshell, this is cracking by trying *every single combination* of password, within some reasonable limit. For example, trying all possible 8 character passwords compromising of lowercase letters, etc. This would mean trying everything from aaaaaaaa → zzzzzzzz by hashing every possible combination and checking against the hash we wish to crack.

A downside of brute-forcing is simply that it becomes unwieldy or impossible with a long enough passphrase. Most modern computers could run through *complex* passphrase of 8 characters within minutes – meaning a totally random 8 character passphrase with letters, numbers, symbols, etc. But trying all possible 16 character combinations could take trillions of years on the same machine. Length gives more possible combinations to try than just adding complexity, so brute-forcing can quickly become out of scope.

That said, within the limitations of processing power, brute-forcing’s advantage is that it can try every possible combination, meaning it’s exhaustive. If you know a system limits passwords to 10 characters, for example, then you know you can try every possible combination in a reasonable timeframe and crack any passphrase you need to. A dictionary attack would most likely miss a randomly-generated 10 character passphrase since it’s not likely to be in the wordlist. But a brute-force attack can find it.

Dictionary Attacks

Unlike a brute-forcing attack, dictionary attacks don't try every possible combination. Instead, they try *likely candidates* from what we call a *wordlist*. Wordlists can be massive, and contain things like passphrases cracked in previous data breaches, dictionary words, or common phrases from media. A dictionary attack works by running through a given wordlist, hashing the potential passphrases, and comparing to the given hash. Sophisticated dictionary attacks can also try variations on those words, like substituting 1 for I or 3 for e. A good wordlist (dictionary) is the key to a successful attack.

The downside of a dictionary attack is the more limited scope of passwords to try. Your attack is only as good as your wordlist and variations, and it won't catch something like a randomly-generated passphrase.

The great thing about dictionary attacks is how much more efficient they can be for finding passphrases than brute-force. Most users don't choose random passwords, but instead use English words or combinations of English words. Password reuse is common, and entropy for user-generated passwords is often low. Dictionary attacks can handle much longer passwords than brute-forcing, since the scope is more limited and there's endless possibilities for building a good wordlist. As well, a dictionary attack still works well on *salted* passwords, since it just need s to add the known salt to the candidate word from the wordlist before hashing and checking.

Rainbow Tables

Rainbow tables are similar to a dictionary attack, but different in one key area. In a dictionary attack, the hashes are computed from the wordlist *on the fly*. You give the password hash, optional salt, and wordlist to the tool, which then computes each possible hash and checks against the given one. With a rainbow table attack, the possible hashes from the wordlist are *precomputed* and stored in a database for easy lookup. Rainbow tables use brute-force combinations or wordlists to precompute a large database of password hashes for quick lookup when needed. Simply feed the given password hash to a database query and see if any existing passphrases are indexed.

Rainbow tables have a large disadvantage in that they don't work with *salted* passwords. Since a salt radically changes the hash output for any given password, you can't pre-compute a list of password hashes to use. Rainbow tables also can require a lot of storage and up-front computation, which is a tradeoff to consider vs. running a live dictionary attack.

A big advantage of rainbow tables is that they are much quicker to look up passphrases. Since the table is pre-computed, feeding a password hash to search gives near instant results. If you know the system's hash algorithm ahead of time, and know that passwords aren't salted, pre-computing a hash table saves time when you're ready to search for a given password hash.

Hashes, Salts, and More

Password cracking is a key part of many successful attacks against systems. Understanding how passwords are stored and cracked gives us good ideas for making *better, more secure* passphrases. Knowing that salts render rainbow tables ineffective, for example. Or that *length* is more helpful than *complexity.* And of course, that *entropy (randomness)* is key. Each type of password cracking technique offers insight into making *better* passphrases – using length, randomness, and proper storage techniques to our advantage against attackers. And, of course, these techniques are very useful for offensive security – breaking into systems for good and testing our own resistance to attacks!