Hash cracking with `hashcat`

The hash operation is a one-way, deterministic function which maps data of any size to a fixed-size value. The resulting value is called a hash. A hash *identifies* the input without giving away *what* the input is. This is useful to store passwords. Instead of storing the password itself, we can store a hash of it. If our database ever gets hacked, the hacker cannot read the password. But, running the original password through the same hash function will result in the same hash. Thus, if a user wants to log in and gives us their password, we can still check if it is indeed the correct password.

Important to note is that hashing is a one-way function. This means that if a hacker acquires a leaked hash, they cannot reverse it to get the original data back. However, they can try guessing what the original data is and run it through the same hash function, similar to how we checked a user's password. If the resulting hash is the same as the leaked hash, the hacker "cracked" the hash!

`hashcat` is a program which facilitates cracking hashes. Downloads and the wiki can be found at https://hashcat.net/. It produces a huge amount of guesses, hashes them, and checks if they match the input hash. This is best done on a GPU, as GPUs are optimised for parallel work. To defend a hash from being cracked, salting and a strong hashing algorithm can be used. A strong algorithm is on that takes longer to compute. This means that each guess will also take longer to hash, slowing down the cracking process.

Usage

The basic flags required are `-m` to indicate the hash **m**ode and `-a` to indicate the **a**ttack mode. Type `hashcat --help` in your terminal to get usage information. It neatly categorises options in tables.

Debugging

Sometimes it is hard to know which guesses `hashcat` is making. Simply add the `--stdout` flag to print all guesses to the terminal without performing the actual attack.

Hash algorithms

The table **Hash Modes** lists the available hash algorithms. The first column is the code used to tell `hashcat` to use that algorithm. For example, to use the MD5 algorithm, use `-m 0`.

Guessing techniques

Of course, without any idea as to what the password might be, guessing will be really hard. In the worst case an infinite amount of passwords must be tested. That is not doable! Some techniques exist to narrow down the guesses. `hashcat` supports these techniques as different **Attack Modes**. We explain the most common ones. See https://hashcat.net/wiki/#core_attack_modes for more information.

Dictionary attack

A dictionary attack, or as `hashcat` calls it: **straight mode**, uses a wordlist. Instead of guessing randomly, the words in this list are used. Basic wordlists can be found at https://github.com/danielmiessler/SecLists/tree/master/Passwords. To use this attack one would type:

```
hashcat -a 0 -m <hash_mode> <hash> <path_to_list>
```

Mask attack

A brute-force attack tries all possible combinations of characters for a given password length.

A mask attack is basically a smarter brute-force attack. It allows specifying different character sets for each index in the string. This includes locking a set to a single static character. A mask attack is thus more specific than a brute-force attack, which is desirable to decrease the amount of possible guesses. It is especially useful if the structure/pattern of the password is known.

The "mask" is what describes the pattern. `hashcat`'s mask syntax is a string of a specific length. Each index in the string is either a static character, or a variable to be filled out with a value from a character set. A variable's syntax is `?<x>` where `<x>` is replaced by the character set's code. `hashcat` provides several **built-in charsets**. Consult the appropriate table in `--help` for a list.

For example, many people use a password that starts with a capital letter and ends on their birth year. If the password is 8 characters long, using built-in character sets, the mask would be `?u?a?a?a?d?d?d?d`.

`hashcat` also supports up to 4 custom character sets, which can be a combination of built-in sets but not of custom sets. It also supports variable length masks with the increment options.

The full syntax for a mask-attack would be:

```
hashcat -a 3 -m <hash_mode> <hash> <custom_charset,...> <mask>
```

Rule-based attack

People with bad password habits often use specific words or names, such as the season or their dog's name, and modify it to fit the password policy. For example, they might replace an `a` with an `@`, add random digits at the end, or capitalise the first letter. A rule-based attack can combine a wordlist and rule set to automatically apply some of these habits to the words.

`hashcat`'s rule syntax is extensive and will thus not be discussed here. Consult https://hashcat.net/wiki/doku.php?id=rule_based_attack for more information. However, we note that `hashcat` comes with several common rule sets pre-installed.

Applying rules is not a separate mode. Instead, it simply extends the list of guesses made via other modes. To use rules, simply add the `-r <path_to_rules>` flag to the command.