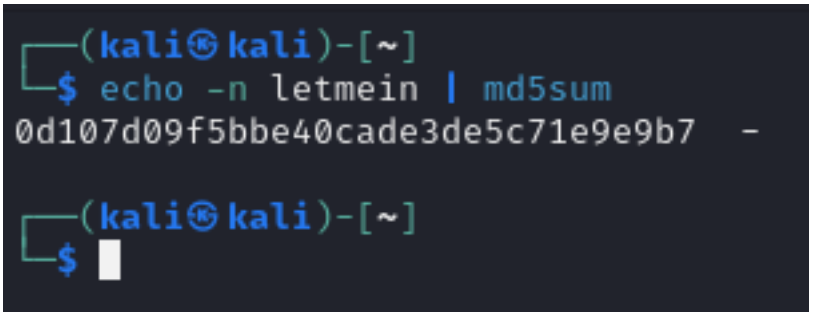
Password Cracking

Let's start by generating a hash to crack against.



This is an extremely simple password. Let's crack it.

Before we begin, hashes are a one-way function. You cannot, I repeat...CANNOT reverse a hash (I've heard people who should know better mention reversing hashes - it doesn't work that way).

You can't decrypt a hash, either. It's not encrypted. The hash is a string of randomly generated letters and numbers that adhere to a hashing algorithm. When you crack a hash, you are not reverse engineering it. You're comparing a known hash to a possible match.

Let's take the hash we just created **0d107d09f5bbe40cade3de5c71e9e9b7**

The ONLY way to figure out the original value here is to have a matching hash. What a password cracker does is either use brute force (which tries every possible combination - way too time-consuming) or wordlists. Wordlist cracking takes a list of known or popular passwords, hashes each list item one by one, and compares that hash to the hash you're trying to crack. If a match is found...there's the password. Because with hashing, the value will always be the same per the hashing algorithm you're using. This is why hashing is effective for IDS and file monitoring systems like Tripwire. They compare file hashes, if there is any subtle change in a file, the hash will be completely different, alerting you that the file has been tampered with.

Back to password cracking.

There are several tools we can use to crack these hashes. There are even online options so you don't even have to install anything.

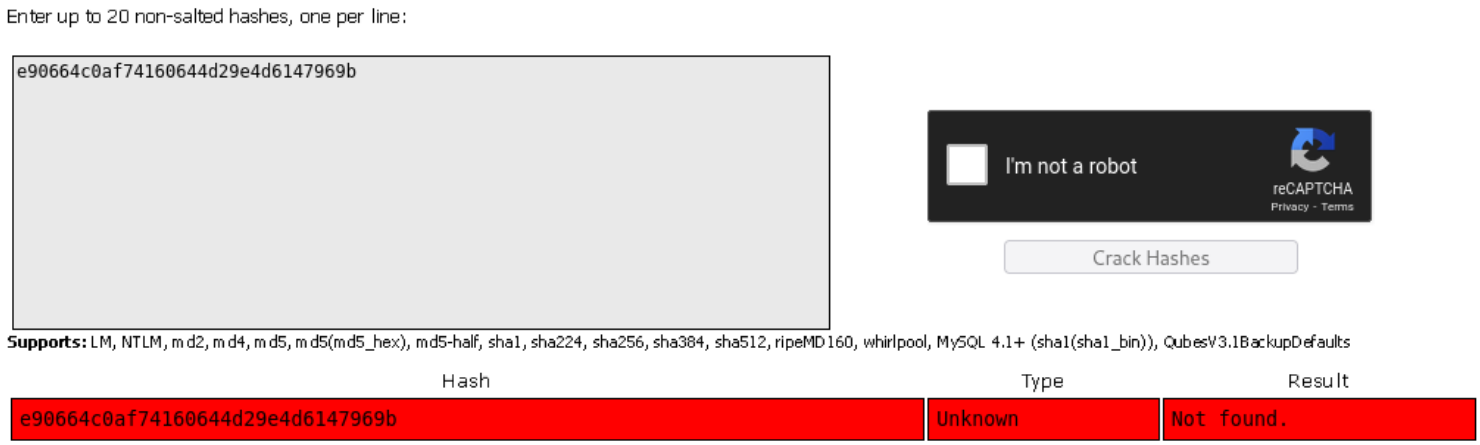
Here we'll use crackstation and see if it is able to determine our password by giving it the hash. It is.

This is a very simple example of cracking a very simple password. Let's try something a little more complex.

```
┌──(kali㉿kali)-[~]
└─$ echo -n Summer2024 | md5sum
e90664c0af74160644d29e4d6147969b  -

┌──(kali㉿kali)-[~]
└─$
```

These are the types of password born of unnecessary password expiration policies (let's not even get started on that). We have a combo of letters and numbers, upper and lower case. Let's try to crack it.



Crackstation was unable to crack this one. That's fine, let's turn to some tried and true password crackers that will give us a lot more options and freedom with our nefarious password cracking techniques.

We can save the hash value to a file, pass it to hashcat and run it against a wordlist called fasttrack (note -m 0 is the hash type we want to crack, the 0 = MD5).

```
┌──(kali㉿kali)-[~]
└─$ hashcat -m 0 sample fasttrack.txt --force
hashcat (v6.2.6) starting
```

The result, after less than a second.

```
e90664c0af74160644d29e4d6147969b:Summer2024

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: e90664c0af74160644d29e4d6147969b
```

So that's password cracking 101, but what about when a password is created using special characters and padding (also very popular once a strong, well-remembered password has been expired).

You're typical password policy is something like this:
• Minimum 12 characters

• Uppercase letters
• Lowercase letters
• Numbers
• At least one special character (symbol)

That said, let's make the previous password adhere to this policy.
Summer2024 becomes **$ummEr2024111**

This password will unlikely be found in a wordlist, but...this is where password mutation comes into play. Using rules, we can mutate wordlist entries. This will allow us to inject common password replacements and add common padding mechanisms. Let's see what this looks like.

Let's hash the new password value and overwrite the value in the sample file with our new hashed value. Then we'll run hashcat again.

```
Session..........: hashcat
Status...........: Exhausted
Hash.Mode........: 0 (MD5)
Hash.Target......: c46e30047efedf1210555aff797f54ef
```

This time we did not get the password. This is becasue it is less common and not found in the particular wordlist we're using (or most other wordlists for that matter).

Enter password mutation. We'll use a rule with hashcat this time.
Note: This is a rule I wrote specificly for this example, but there are built-in rules that are highly effective at cracking some pretty complex passwords.
The rule will replace all S values with a $, it will then change evey lowercase character to uppercase (one at a time), and finally append 111 to the end of each word. It will perform these actions for each entry in the wordlist.

Let's run hashcat with the -r option and pass the rule.

```
┌──(kali㉿kali)-[~]
└─$ hashcat -m 0 sample fasttrack.txt -r crack.rule --force
hashcat (v6.2.6) starting
```

Even though the password was not in our initial wordlist, mutating it with a rule still allowed us to crack it (and within a second)!

```
c46e30047efedf1210555aff797f54ef:$ummEr2024111

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: c46e30047efedf1210555aff797f54ef
```

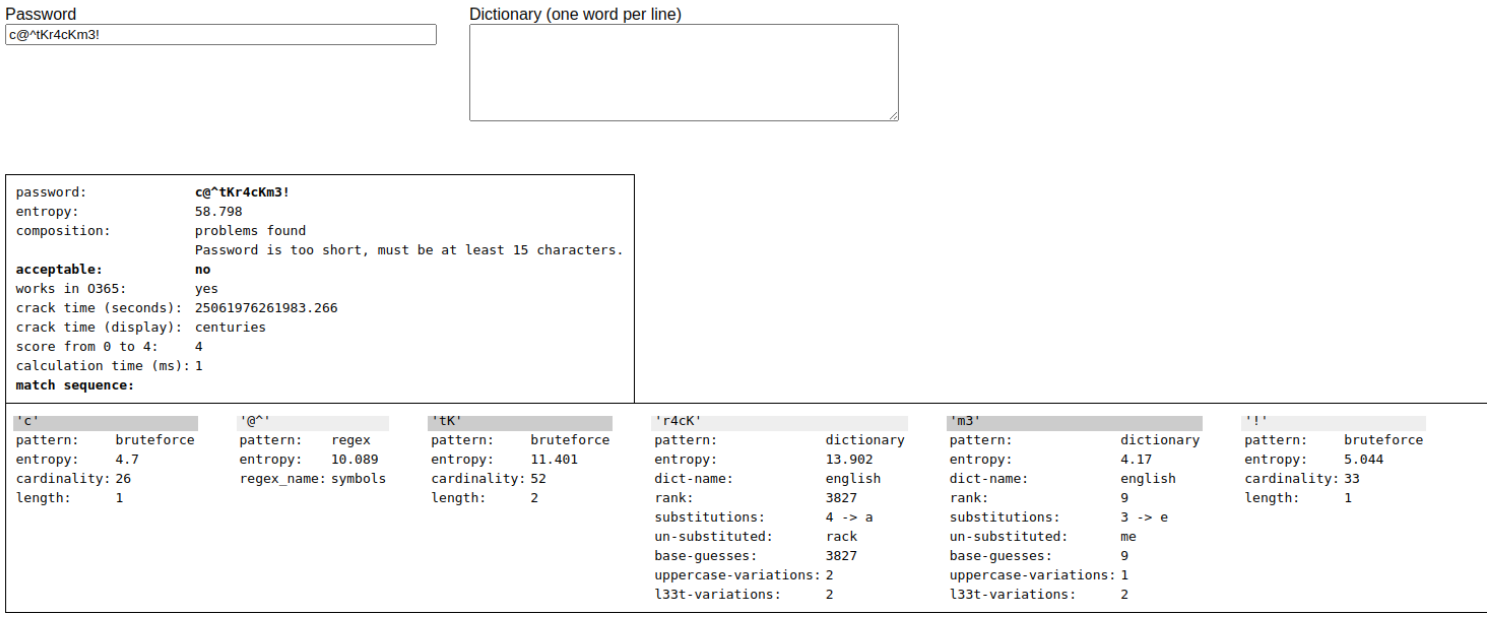
Sometimes this process is trial and error, also enumeration of password policies greatly helps in creating rules that will likely be effective.

I made a comment earlier about expiration policies. Let's demonstrate how these undermine password security.

Jane starts work at a new company. She's given the policy above and told to create a strong password.

She selects **c@^tKr4cKm3!**

Let's pass this to a strength checker (I really like Cygnius, because it takes entrophy and patterns into account).



This is a great password! It's considered a little short, but still very strong, will not be found in a wordlist, and would be very difficult to crack using even an extremely complex rule. Also, you won't find this password on a darkweb list or leak, becasue it very likely will not be compromised. THERE IS NO LOGICAL REASON TO CHANGE THIS!

But.. 120 days in (sometimes less), Jane is forced to change this password. What?? Why?? Ok, so she needs to come up with something else strong that adheres to the policy, more importantly, something she can remember.

Now she comes up with **@nT1cracK012**

This still seems ok, but...

Password

@nT1cracK012

Dictionary (one word per line)

```
password:              @nT1cracK012
entropy:               32.345
composition:           problems found
                       Password is too short, must be at least 15 characters.
acceptable:            no
works in O365:         yes
crack time (seconds):  272718.95
crack time (display):  5 days
score from 0 to 4:     2
calculation time (ms): 1
match sequence:
```

```
'@nT1'
pattern:               dictionary
entropy:               14.387
dict-name:             english
rank:                  2678
substitutions:         1 -> i, @ -> a
un-substituted:        anti
base-guesses:          2678
uppercase-variations:  2
l33t-variations:       4
```

```
'cracK'
pattern:               dictionary
entropy:               11.788
dict-name:             english
rank:                  1768
base-guesses:          1768
uppercase-variations:  2
l33t-variations:       1
```

```
'012'
pattern:               sequence
entropy:               3.585
sequence-name:         digits
sequence-size          10
ascending:             true
```

One policy change has completely undermined the security of the password already (imagine how easy this will be to crack in a year! The expiration policy is helping us tremendously).

Ok, last scenario. Jane survived a year with the company...her password strength has not. She is forced to make yet another unnecessary change.

Jane throws her hands in the air and shakes her head. Whatever, just put something in that meets the requirements.

$ummEr2024111

I rest my case.