2022-04-29

Ray Morris

sensei@securityguy.dev

**Reversing Hashcat**

How I Cracked Passwords to Keep Them Safe

Table of Contents

1. Introduction
2. Background
   1. Problem intro
   2. Scope of the problem
3. Existing Defenses
   1. Hashing
   2. Rate Limiting
   3. Two Factor Authentication
   4. Existing filters
4. New Approach - Reversing Hashcat Provides Definitive Answers
   1. Idea - reverse Hashcat attacks
   2. Crackers publish their techniques
   3. Impact: Simulation of attacker prevents attacks
5. Solution Implementation Methods - Reversing Each Attack
   1. Dictionary attack
   2. Rule attack
   3. Mask attack
   4. Hybrid attack
6. Filter Evaluation
   1. Methods
   2. Results
   3. Discussion
7. Risks, Limitations, and Costs
8. User Interface Developed
9. Secure Password Generator Implemented
10. Summary and Conclusions
11. Future Work
12. Developed Source Code Repositories
13. Acknowledgments
14. References

# Introduction

Weak passwords have led to breaches of President Trump’s Twitter account not once, but twice (Carolina, 2018). Poorly chosen passwords led to the breach of the Canadian Revenue Agency, and contributed to the spread of the Purple Fox malware (Odogwu, 2021; Specops, 2021). While some commentators have suggested that organizations should switch to smart cards or other types of authentication, they have their own drawbacks and most organizations continue to use passwords.

Currently available passwords meters are ineffective for ensuring that passwords are actually resistant to attack, since they do not measure the types of attacks carried out by real attackers. During the first phase of this project, existing password meters were used to test passwords rated as strong using real crackers performing real attacks. Currently available meters performed only slightly better than chance, about three to four percent better than coin flipping to determine if a potential password was strong or weak.

Existing password meters are not effective for ensuring passwords are actually resistant to attack. This is because they do not generally measure anything directly related to the types of attacks actually used by real attackers. In the first stage of this project, passwords rated as strong by existing passwords meters were tested with real password crackers performing real attacks. Existing meters performed only slightly better than chance, three to four percent better than calling a potential password “strong” or “weak” based on a coin flip.

A new and significantly more effective method is required, one that is compatible with the existing installed software base. This new, more effective method is presented here. This new method is based on measuring the resistance to specific, well-defined attacks actually used by attackers.

# Background

Security incidents in the news recently involving passwords include a researcher guessing President Trump’s password. According to court testimony, Victor Gevers gained access to Mr. Trump's Twitter account in 2016 by guessing the password, "yourefired." In 2020 he did it **again**, guessing the password “maga2020!” on his fifth try. Any attacker who guessed “maga2020!” and was therefore able to post as Mr. Trump could have escalated international tensions with a message that would have appeared to come from him. Another attack in 2018 targeted several members of the Northern Ireland Parliament after guessing weak passwords (Carolina, 2018). In addition, companies such as Alibaba, Dunkin' Donuts, and Magento have suffered breaches because of weak passwords (Odogwu, 2021).

It has often been said that organizations should switch away from passwords to smart cards or other forms of authentication. Although smart cards have been available since the 1980s, passwords are widely used (Jacquinot 2022). The drawbacks of transitioning to smart cards have left organizations still using passwords. Even the National Security Agency (NSA) and Director of National Intelligence web sites direct recruits to a password login form (IntelligenceCareers.gov). Over 35 years since the introduction of smart cards, passwords are still the standard and therefore need to be secured. Even where systems administrators would like to replace passwords with something else, the organization often does not create and maintain the software they use. Rather, they use COTS software and if the software in place uses passwords for login, passwords must be used.

Preventing breaches therefore requires strong passwords. Weak passwords, including re-used passwords, can lead to breaches costing millions of dollars. With the advent of cyber warfare, weak password security may even cost lives. Existing password policies such as requiring uppercase and lowercase letters are demonstrably not solving the problem because passwords such as “Password1!” and “Solarwinds123” continue to lead to breaches. A new approach is required - but one compatible with incumbent password-based systems.

Leaks can expose many millions of passwords. For example, the LinkedIn breach affected at least 6.5 million users (Kamp 2012). The Yahoo breach, over 1.5 billion (Trautman and Ormerod). In total, Haveibeenpwned catalogs nearly 12 BILLION leaked passwords and hashes, and over 613 million unique passwords available as plaintext.

## Asset and Threat Model

Passwords are used in many contexts. Some, such as large web sites, involve millions of users who have only a passing connection with the organization. The organization may have the full source code of the systems used for authentication on the site, and may be able to make any changes required. As an example, they may be able to choose a strong, slow, hashing algorithm to store passwords. However, these sites are limited to providing services via a standard web browser - they cannot reasonably require all users to purchase and register any particular type of smart card or other authentication token, cannot control what software the users use on the client side, etc. These organizations require solutions which are compatible not only with all standard web browsers, but also users’ expectations based on de facto standard practices that web sites normally use. Friction in the registration process and password selection may lead to lost sales or revenue.

Others, such as corporate Windows networks, have a smaller number of users - hundreds to thousands. They may have some control over the client side, but allow authentication to a variety of software they do not control - Windows, Salesforce, SAP Concur, Jira, Office 365, etc. In order to implement these software products, these organizations must work with the existing authentication capabilities of all of these existing software products. They cannot change the fact that Windows caches passwords with comparatively weak DCC2 hashing, but must instead ensure passwords are strong enough to resist attack despite the weaker hash algorithm.

In the case of large web sites such as LinkedIn or Yahoo, password hashes may be exposed in bulk. On a Windows corporate network, one or a small number of administrator passwords may be cached on each computer. By cracking this one single hash, an attacker may gain administrative access across the entire domain.

This project seeks to define and implement a solution that protects both types of scenarios, with a particular focus on the corporate environment, which includes COTS software, such as Windows. A solution that creates passwords strong enough to be secure even when hashed with DCC2 or the MD4-based NTLM algorithms used by Windows will also be secure when hashed with a stronger algorithm by a web-based service.

Threats envisioned therefore include an intruder who has acquired either DCC2 cached hashes from Windows workstations, NTLM hashes from Active Directory, or large numbers of other hashes from web properties.

# Existing Defenses

## Hashing

Passwords are frequently hashed and stored using algorithms such as MD5, SHA-1, or SHA-2. This prevents casual observers from seeing the plain password if they get access to the database. However, these hashes can be broken, or "cracked," by knowledgeable attackers when passwords are not sufficiently strong. For example, the Hashcat program running an Nvidia RTX 3090 GPU can try 9.7 million guesses per second for SHA-256 hashes (Epixoip 2016, Croley 2020). This means that if the user uses any of the top 97 million passwords, the attacker can crack the hash in ten seconds or less. Similarly, attackers can make lists of words such as the company name, email addresses, etc. Using these wordlists and readily available rule lists, Hashcat can generate and test millions of guesses per second. Hashing, therefore, does not protect passwords. Other software for this purpose includes John the Ripper, the first well-known software for cracking passwords. Slower algorithms can make it take longer to make millions of guesses, but tcannot protect against passwords such as "companyname123", "solarwinds123", or "Password1!" and other passwords that take less than a million guesses.

On hashmob.net, we can see that for many of the leaked or breached lists, the crackers have cracked over 90% of the password hashes, often tens of thousands of passwords for a given company or government entity. I submitted a list of 47,000 real user-chosen passwords and the crackers of Hashmob cracked over 99% of the passwords. This demonstrates that with existing solutions, the passwords chosen are not resistant to cracking.

## Rate Limiting - Account Lockout After Failure

Many systems use failure thresholds which will lock an account for a period after a number of incorrect passwords are attempted. It should be noted this approach only applies for online attacks, in which the attacker blindly submits guesses directly to the system they wish to access. It does not apply to offline attacks for which the attacker has password hashes. These include scenarios such as Yahoo, Linkedin, and any Windows system, which stores a hash of all passwords used in order to facilitate offline login. This means that when a domain administrator or system administrator logs in to a Windows system, a hash of their password is left behind on every system they access. This also applies to service accounts for software such as Solarwinds which has privileged access to every computer in the enterprise. By using their own computer to crack the hash, attackers bypass any lockouts configured.

Additionally, lockouts can be partially bypassed even during online attacks, when larger organizations are attacked. The attacker can simply do a breadth-first attack rather than depth-first. Consider the standard Windows policy of five failed attempts per user account in 15 minutes. If the company has 4,000 users, the attacker can try 4,000 X 5 guesses, every 15 minutes. This is 80,000 guesses per hour. Running a script overnight the attacker can therefore attempt nearly a million different passwords. A weak password such as “Spring2022” is not secure under these conditions.

## Two Factor Authentication / Multi-factor

Because two-factor authentication provides some degree of additional security in some situations, one might think that it negates the need for strong passwords. However, the most popular 2FA systems, such as TOTP and HOTP, which use a 6-digit code, fall to the same arithmetic as lockouts. With a million or more guesses made overnight, it is likely that the correct 6-digit code will be hit by chance. This is particularly true because the systems need to allow not only the correct code, but also allow the client and server to be out of sync by at least one in each direction. This means there are always at least three codes that will be accepted. Additionally, there is ample research available about various ways around different 2FA systems, such as "SIM jacking." See, for example, Konoth, Veen, and Bos (2016). Two-factor authentication is complementary to strong passwords. If the password is not strong, the system is left with essentially single-factor authentication, where the remaining factor may be as weak as a 6-digit number.

## Existing filters

Efforts have been made to develop software filters that either prevent users from choosing weak passwords, or at least provide a password meter informing the user of the relative strength of a candidate password. For example, Microsoft Windows can require that a password contain three of these: an uppercase letter, a lowercase, a digit, or a special character. This means that "gfuivdfagfyugvoadu" is not accepted, but "Password1" is considered sufficiently strong. Another example is a password meter created by Solar Designer, who is also the author of John the Ripper. This implementation is called "passwdqc." As described in the passwdqc README file, the tests are primarily based on requiring uppercase letters, lowercase, digits, and punctuation, or "special characters." Zxcvbn is a similar meter by Dropbox, with a more complex calculation of password strength. Another of this type is libpwquality, which also checks for palindromes, sequences of the same character repeated, and certain other tests. An essential problem with this approach is that password cracking is in no way limited to trying only one class of characters. Attackers can try "Password!" just as easily (or more easily) as they can try "dallastexas". Capitalizing the word doesn’t prevent it from being guessed. The effectiveness of these filters was tested, with details provided below. Test results indicated that other than identifying very short passwords, these types of filters are only slightly more effective than randomly choosing to call a password "strong" or "weak" based on a coin flip. The full testing methodology is elucidated below.

OpenPasswordFilter and others in its class operate from a "deny" list, rejecting perhaps the 10,000 most popular passwords. As mentioned previously, an attacker can try over 9 million times per second. Therefore, even blocking the top 1 million passwords means the system blocks only those that the attacker is likely to test in the first 112 milliseconds. After the first 112 milliseconds, the attacker will have moved on past the top million most popular. A new method is required.

# Solution Approach - Reversing Hashcat Provides Definitive Answers

A new approach tested here is not based on heuristics such as testing for uppercase letters, but rather actually tests whether a candidate password will indeed get cracked by methods actually used by password crackers. The Hashcat and John the Ripper web sites provide information about how these tools are used to crack hashes. The documentation shows how the tool cracks hashes and under what conditions the tool can in fact crack a given password. Additionally, each year at Defcon, there is a password cracking competition. Each of the last three years, Team Hashcat has won. Therefore, they are arguably the best password crackers in the world. Each year, Team Hashcat does a write-up on their wiki explaining how they cracked the hashes (Team Hashcat, 2021). Write-ups are also available from other competitions. This means defenders can see just how hashes get cracked. We can therefore defend against the actual attack methods, such as rules attack and combinator. This is a fundamentally different approach than trying to estimate how likely it might be to get cracked, based on characteristics which may or may not correlate with any of the attacks actually used. The new system proposed and developed in this paper is designed to check in a definitive fashion whether a given password candidate would actually be cracked using each specific attack method and specific data sets actually used by password crackers. In this way, the new tooling can make determinations such as these:

* The candidate password will be cracked by using the NSA64 ruleset and 1 million wordlist.
* The candidate password will not be cracked by using any of the 52,000 known rules, applied to the 11 million entry wordlist.
* The candidate will not be cracked via dictionary with any dictionary known to Hashmob (554 million entries)
* The candidate will be cracked via dictionary attack with most dictionaries of 1 million or more, because it is in the top 500,000 most-used.

These are fundamentally different and much more precise findings than those that can be achieved by simply checking uppercase and lowercase characters, punctuation, or length. In essence, this project seeks to simulate an attacker. If an attacker can be simulated well, that will determine whether or not a given proposed password would be cracked. A sufficiently accurate simulation could therefore potentially prevent the 81% of data breaches that are caused by poorly-chosen passwords, according to the Verizon report (Verizon, 2017). It may be noted here that while this project focuses on offline attacks, which can involve trying billions of potential passwords, strong passwords also protect against online attacks, which may allow attackers to try hundreds or thousands. That is, the set of passwords sufficiently strong to resist offline attack is a proper subset of those strong enough to withstand online attack. The proposed method will therefore provide protection from both types of attack to the extent it ensures passwords are strong enough to resist offline attack.

In order to allow this simulation to be done quickly enough to provide real-time feedback to the user, the proposed solution takes advantage of full knowledge of the actual proposed password in order to make the simulation run far more quickly than the attack would run without this knowledge. For the simplest example, consider the dictionary attack. An attacker might try 10 million possible passwords from a list. In order to simulate this to see if a given candidate password would be found, we need only check to see whether the user’s known choice of password is in the list. While an attacker would need to try all 10 million passwords, we as defenders only need to perform a single lookup to determine if the proposed password is in the list. Appropriate data structures allow this to be done extremely quickly. Expanding on the idea that looking up the (single) candidate password in the list is the inverse or reverse of trying every item in the list, this project develops the inverse or reverse of each of the most important attacks that hashcat and John the Ripper can perform. The project has, in a sense, attempted to reverse or invert the Hashcat attack tool. Because it can be seen as Hashcat in reverse, the new system has been named Password Dog.

After reviewing the write-ups and conducting extensive conversational interviews with members of Team Hashcat and another group they referred the researcher to, Hashmob, three types of attacks were identified as the primary attacks used by crackers. Several weeks of discussion with top password crackers and testing were conducted in order to fully understand the attacks and how they are used. Inverted cases of each of the three were then developed. The three primary attack methods used by champion crackers were dictionary attacks, rule-based attacks, and mask attacks. Additionally, the hybrid attack was partially inverted. The new solution checks each type of attack and returns the number of tries needed for the most efficient attack as the overall strength of the candidate password. Short passwords with length less than 8–10 may also be brute forced, particularly when faster hashes are used, such as MD4/NTLM, MD5, SHA1, or SHA2.

# Solution Implementation Methods - Reversing Each Attack

## Checking Brute Force and Digit Brute Strength

For efficiency, before performing more advanced checks, the candidate password is first checked for total length. The number of attempts required for brute force is calculated. If the number of required attempts is less than 108, the password is considered weak and processing terminates. A similar check is done for passwords that consist only of digits. This very fast client-side check allows instantaneous feedback to the user with negligible processing cost.

## Reversing the Dictionary Attack

Perhaps the simplest attack is to try the 1,000 most commonly used passwords, or the 1 million most commonly used, or any number based on the available resources and the speed at which the hashing function can be run. To reverse this attack, bloom filters were made in order to allow quick lookups to see if the candidate password is in the top million, top 11 million, or a full list of 554 million known passwords. While the contents of different "top" lists differ, research found much of the content is the same across lists, with the order being different and the content differing in some of the least-used passwords on the list. The hypothesis is that if a candidate password is not in our list of **eleven** million most commonly used, it is likely not in the attacker’s list of **one** million most commonly used. Conversely, if it is found in our list of 11 million, it may well also be in an attacker’s list of one million or ten million.

Bloom filters were created for 11-million, 184-million, and 554-million entry lists that may be found on sites used by password crackers. Bloom filters are a data structure that allows the system to quickly check whether a given candidate password is in the list. By being multi-dimensional and probabilistic, the bloom filter requires less space than storing the full list. For example, the bloom filter for the 554-million entry list is 1,400 MB—less than three bytes per entry. The bloom filters were generated with the BloomFilter Python module. The strength of a password candidate against dictionary attacks is computed based on the size of the dictionary (list of common passwords) in which it was found.

## Reversing the Rule Attack

The rule attack expands on the dictionary attack and is a primary method for password crackers. Tools such as Hashcat and John the Ripper allow crackers to define a set of *rules* which modify each item in the wordlist. For example, rules might add year numbers to the end of each item, turning MyPassword into MyPassword1985, MyPassword1986, MyPassword2021, etc. Password crackers analyze the effectiveness of different rules and assemble collections of the most effective rules, such as the well-known "best64" collection included with the Hashcat tool. The first rules in this list are as follows:

* : (no change)
* R (reverse the string)
* U (upper case)
* T0 (toggle the case of the first character)

Other rules delete characters, so that "BaltimoreRavensXXXV" in the wordlist will generate and crack "BaltimoreRavens", for example. To reverse this, the project needs to perform two steps:

1. Determine which types of rules would match the candidate password to something in the wordlist
2. Determine whether such rules exist in rulesets likely to be used by attackers (rules known to be effective)

To perform the first step, the project leverages pre-existing work in the Password Analysis and Cracking Kit (PACK) toolkit by iphelix (Kacherginsky, 2019). After some minor preparatory work on the string, such as handling "leet" substitutions, the rulesgen.py tool included in PACK uses a spell checker engine to find which entries in the word list are most similar to the candidate password. For each item in the list, it computes the rule necessary to make the list item match the candidate password. The code of the new scoring tool calls functions within rulesgen.py to perform this step.

This list of rules, which *would* crack the password, is then checked against a large list of rules that crackers might use. The list used for testing is a compilation of many earlier rulesets. Containing 52,014 rules, the OneRuleToRuleThemAll set is a rather large set and includes the rules from many sets that attackers might use, including best64 and NSA64. It is hypothesized that attackers are generally more likely to use rules that are more effective in testing. The list of 52,014 is ordered based on effectiveness as tested by Kacherginsky. The hypothesis is that rules that appear higher on the list (those that are more effective) are those that attackers are most likely to try. The score on this test is calculated based on that. The score is reported as the total number of attempts that an attacker would need to make before getting the correct password, if they tried them in the order of OneRule. It is possible that attackers will try them in a similar order, if not exactly the same order.

## Reversing the Mask Attack

The mask attack is the one attack that does directly involve character classes such as uppercase, lowercase, digits, etc. It uses specific *patterns* of characters, such as digit: digit: digit: digit (a four-digit pin or year). The mask attack works by analyzing lists of passwords actually chosen by human users to determine certain patterns. For example, the following patterns are common in the RockYou list, with a manageable keyspace for each pattern:

?d?d?d?d

?l

?d

?d?d?d?d?d?d

?d?d?d?d?d

?d?d

?d?d?d

?l?l

?u

[ ?d = digit, ?l = lowercase, ?u = uppercase ]

Knowing that people frequently choose passwords consisting of four digits and knowing that they could try all possible four-digit passwords very quickly, ?d?d?d?d is a rule crackers are likely to try early in the cracking attempt. The mask ?l, representing a single lowercase letter, requires only 26 operations to cover the keyspace. The new tool computes what mask would be needed to generate the candidate password and then checks the position of that mask in a large, ranked list of masks. The rank is again based on the idea that crackers will try the best masks first—those masks that match the highest number of actual human-chosen passwords in the least amount of time. A formula derived from quadratic regression gives the approximate number of mask-generated attempts an attacker would most likely need to try before getting this candidate via a mask attack. The mask list used by the tool includes those needed to produce up to 1022 strings, which would require an unreasonable amount of resources to exhaust, and such passwords would be considered strong against mask attacks.

## Reversing the Hybrid Attack

An attack which was seen less often was the hybrid attack. The hybrid attack combines the dictionary attack with the mask attack. It takes as input a wordlist and a mask pattern, such as:

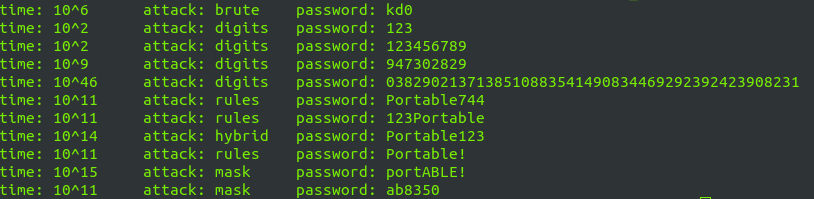
hashcat -a 6 rockyou.txt ?d?d

This will try appending each two-digit number to each item in the rockyou list. As an example, if the source wordlist contains "CentralHigh," this hybrid attack would find "CentralHigh96." Note that the same effect can also be achieved with a rules attack. Because the hybrid attack was only seen being used to append or prepend digits, this case of the attack was inverted for the analysis tool. It currently implements inverted hybrid attack only for appending or prepending digits. Scoring is straightforward—the number of digits implies the average number of attempts needed to come up upon the correct digits for the numeric portion. For the string portion, the size of the wordlist that contains the base string gives the scoring factor. These two are multiplied to get the total number of attempts required, and thus the score.

## Combined Score

In order to be conservative about leaning toward safer passwords, the total score for a candidate password is reported based on the attack that would most quickly crack it. That is, the final score is the lowest score received on any attack. The score is reported log10, or order of magnitude, in order to keep the scores to manageable numbers, generally in the range of <7 (very weak) to 22 (very strong). Scores are reported, rounded to the nearest integer. Along with making the reported scores easy to use, this also highlights that the scores indicate whether cracking the password would be expected to take days, months, or decades. It does not report with a precision that would imply it is measuring the exact number of seconds, nor does it appear to distinguish between 14 and 16 days. If any attack is able to crack the candidate password in less than 108 tries, the password is reported as weak and further analysis of other attacks is bypassed in order to conserve resources.

This screenshot shows the raw output of Password Dog in command line mode:



# 

It also provides a JSON API for web-based and other user interfaces:

{"score": 8, "attack": "dictionary"}

By using this output, we can determine which attack would be most efficient against the candidate password and how many attempts it would take.

# 6. Evaluation

## Evaluation Method

The goal is to predict whether a candidate password will likely be cracked by password crackers. Therefore, to obtain ground truth, a corpus of hashed passwords was provided to skilled crackers. The determination was then made of which passwords were *actually* cracked, and this was compared to the filter’s prediction of which ones were likely to be vulnerable. The accuracy rate was measured by calculating how well the filter predicted whether a given password would be cracked.

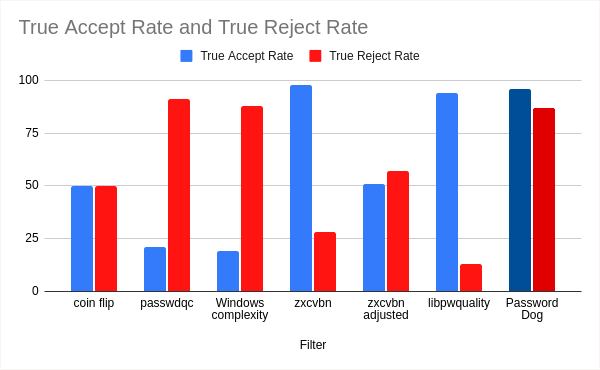
Two members of Team Hashcat, three-time winners of the password cracking contest at Defcon, were contacted. They assisted by trying to crack the password hashes using their standard attack methodology. They also referred me to a community of password crackers, Hashmob, which shares some of the same members and has members with a similar level of skill, according to the members of Hashcat. Tests were conducted both with Hashmob as a group and with another member of Team Hashcat individually to determine which passwords would be cracked.

Measuring was somewhat difficult, requiring multiple attempts to find a solid data corpus for the measurement. Initially, hashes of 47,700 passwords randomly selected from a breach were provided to the skilled crackers. All but one of the hashes were cracked. Clearly, this would not provide a good basis of truth to measure the predictions of the filters regarding which passwords were strong. Adjustments needed to be made to have a corpus with some passwords that would not get cracked. It was determined that many of these passwords were less than seven characters. It can be calculated that all possible passwords consisting of ASCII characters of length 7 can be attempted in less than 2 ½ hours for hashes such as SHA-256 (Epixoip 2016, Croley 2020). It is therefore trivial to determine that such candidates are breakable via simple brute force. A proper measure would consist of passwords at least eight characters long. Additionally, some of the passwords used for the initial trials were already in the database used by at least some of the crackers, so they could crack them via dictionary attack only because they had seen that leak before. This did not accurately represent whether fresh passwords from a new leak would be cracked.

To test the new filter 4,000 properly, actual user passwords were sampled from the Clearvoice breach of 2021. This breach was selected for two reasons. The passwords were leaked in plain text rather than as hashes, making all of the passwords available rather than only the passwords that had been cracked. The passwords were not, however, in the database of Hashmob.net, meaning participants could not crack the hashes by simply running a dictionary attack using the databases expected to be available to them. This simulated a new leak. Passwords were chosen randomly using Gnu *shuf*, after selecting only passwords greater than 9 characters. 4,000 samples were chosen randomly. To simulate an intrusion into a Windows network, the passwords were hashed using the DCC2 hash algorithm, also known as mscache. This is the algorithm used by Microsoft Windows to store credentials on each machine in order to facilitate offline login. This is a moderate-speed algorithm suitable for "average" attacks, neither so fast that inefficient attacks can be used with abandon, nor slow enough to frustrate attacks that are likely to be used in practice against other hash types (Epixoip 2016, Croley 2020).

## Evaluation Results - *Password Dog* Filter

The accuracy of the new filter was compared to that of four existing meters. Libpwquality is an open source password filter that integrates with the Linux PAM system. Passwdqc is a meter authored by Solar Designer, the author of John the Ripper and a pivotal figure in establishing modern password cracking techniques. The Windows complexity test is the default rules provided by Microsoft Windows - must have at least three of: [uppercase letters, lowercase, digits, special characters] Zxcvbn is a password filter developed by Dropbox. This was tested with the default threshold as suggested in the documentation, as well as with a higher threshold to be more strict. Finally, for context, the chart compares the null case—a simple coin flip, which calls a password strong or weak based on randomly flipping a coin. The coin flip has an accuracy rate of 50%.

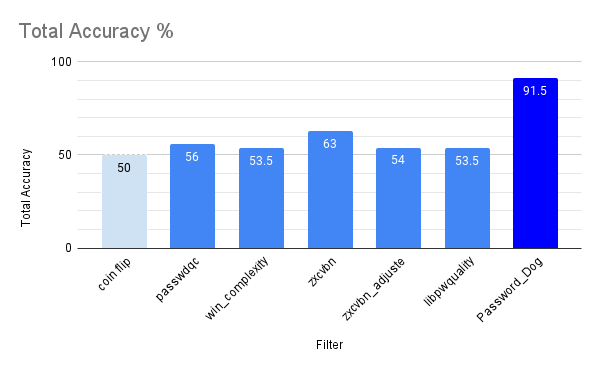


This figure shows in blue the true acceptance rate, that is, the percentage of passwords that did not get cracked that were rated "strong" by the filter/meter. This measures the filter’s ability to accept a password that is in fact strong. In red, it shows the true reject rate-the percentage of actually weak passwords that were correctly rejected by the filter. These blue bars show the filter’s ability to reject weak passwords. For precise numbers, the same data is here shown in table format:

**Filter Accuracy**

| Filter | True Accept Rate | True Reject Rate | Total Accuracy |
| --- | --- | --- | --- |
| coin flip | 50 | 50 | 50 |
| Passwdqc | 21 | 91 | 56 |
| Windows complexity | 19 | 88 | 53.5 |
| Zxcvbn | 98 | 28 | 63 |
| zxcvbn adjusted | 51 | 57 | 54 |
| Libpwquality | 94 | 13 | 53.5 |
| Password Dog | 96 | 87 | 91.5 |

The different filters varied significantly in the types of correct predictions and types of errors. Libpwquality and Zxcvbn accepted most of the strong passwords, while incorrectly accepting most of the weak ones as well. Conversely, passwqc and the built-in Windows rule correctly rejected many of the weak passwords, at the cost of also rejecting most of the strong passwords. In order to make comparison easier, the following chart shows the total accuracy rate by stacking the correct acceptance bars with the correct rejection bars.



Measured by total accuracy rate, Zxcvbn performed the best of the pre-existing meters. It has an accuracy of 63%. The new filter, Password Dog, scored 91% total accuracy.

The accuracy rate was also calculated in the other direction, by computing the percentage of predictions that were correct. This was computed versus two groups of attackers, starting with the predictions made by Password Dog and determining the accuracy rate for those predicted to be strong and those predicted to be weak. The statistics were calculated based on how well Password Dog predicted whether Hashmob would collectively crack them. Separately, the calculation was also done based on whether one well-known member of Team Hashcat, Sam Croley (Chick3nman), cracked them. The results are similar by either measure.

### Vs Hashmob:

Of 2,519 allowed by Password Dog, 2318 (92%) withstood attack by Hashmob.

Of 1481 Password Dog rejected, 1377 (93%) were indeed cracked by Hashmob.

### Vs Croley:

Of 2,519 allowed by Password Dog, (93%) withstood attack by Croley.

Of 1481 Password Dog rejected, 1311 (89%) were indeed cracked by Croley.

A note: Mr. Croley chose to discontinue the attack after two days - an overnight run of his standard attack and then some time manually tuning and attacking. After that, the rate of cracking dropped off significantly. It should be noted that Mr. Croley *could* have chosen to continue manual attacks and perhaps recover more. The figures stated indicate what Mr. Croley did in fact crack - there may be others he *could* crack given sufficient time and motivation.

## Results Discussion

Existing filters had total accuracy rates ranging from 53.5% to 63%. By comparison, randomly calling a password "strong" or "weak" based on a coin flip has an accuracy rate of 50%. The filters, then, performed 3.5 to 13 percentage points better than random. The best of the existing meters, Zxcvbn, scored 13 percentage points better than the null filter, a coin flip. The developers of Zxcvbn stated it was originally to protect against online attacks rather than the more challenging offline attacks tested here, though the documentation also provides settings for offline attack protection (Wheeler 2016).

Measured by total accuracy rate, Zxcvbn performed the best of the pre-existing meters. It has an accuracy of 63%, or 13% better than a random coin flip. The new filter, Password Dog, scored 91% total accuracy vs. a champion password cracker and 92% vs. Hashmob. This is a significant improvement over previous methods.

Passwdqc accepted only 21% of the strong passwords, the passwords that did not get cracked. The Windows complexity rule accepted only 19% of uncracked passwords. This reduces the usability of these systems. In contrast, Password Dog accepted 96% of the strong passwords, a much better rate of accepting passwords that are in fact strong. Password Dog correctly rejected 87%–89% of the passwords that were crackable. (87% as measured against Hashmob, 89% as measured against Croley). It could therefore potentially prevent 87%–89% of breaches and incidents caused by weak passwords if organizations used Password Dog or another filter built on these principles.

As discussed above, all short passwords are weak; all passwords less than eight characters can be cracked in only 2 ½ hours. Therefore, rating these short passwords as weak is trivial. For this reason, testing was done with passwords of length 10 or higher. The percentage scores for all filters may well be higher if tested with short passwords. For example, Password Dog will correctly label passwords as weak 100% of the time, for passwords under eight characters.

It may be noted that the results of testing Password Dog against Hashmob and against Croley, who is not associated with Hashmob, are quite similar. In fact, of the 1480 passwords cracked by Croley, 96% of those were also cracked by Hashmob. This indicates that it is possible for Password Dog to predict that a password will or will not be cracked by a competent attacker without specifying which attacker; competent attackers tend to crack the same passwords. Interviews with professional and champion password crackers affirm this idea, though hard data was not collected beyond comparing Hashmob to Croley.

# 7. Risks, Limitations, and Costs

The results do show that there are some passwords that Password Dog rated as strong but which actually did get cracked. A potential avenue of future improvement would be to study those that "slipped through the cracks" and determine how to better identify them. Some potential strategies have been identified in the Future Work section below. There is some risk that people could use passwords rated as strong, then if hashes are leaked, some of those passwords do actually get cracked. For this reason, Password Dog, like any other filter, MFA, or other strategy, should be treated as a way to *reduce* risk from poorly-chosen passwords, not a foolproof shield to eliminate all such risk.

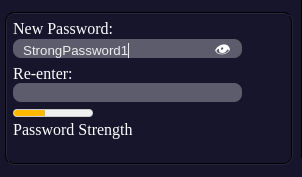
Similarly, Password Dog can only help prevent choosing poorly-chosen passwords. Storing passwords securely is outside the scope and must be handled by other systems. This will often include systems to be sure secrets are **not** stored in inappropriate storage, such as public source code repositories. Commercial solutions are available from a number of companies to provide safe storage. Additionally, Password Dog currently ranks the strength of a password in a global context rather than a personal context. It does not currently check whether the password includes elements derived from the user's name, handle, or email address, for example. Adding a per-user wordlist is an item for future work. Per-site lists could also be readily added.

Any software that handles passwords in any fashion necessarily carries a risk that the software could contain a vulnerability of some kind which could leak passwords. In particular, the communication channel between the user interface used by the user and the Password Dog code needs particular attention to security. It is recommended that Password Dog be deployed in an appropriate sandbox or container, after an appropriate security audit of the implementation plan and the code itself.

The computing resources needed for Password Dog deployment are minimal. All testing was carried out on a nine year-old laptop, equipped with a third generation Core i5 and a magnetic hard drive (non-SSD). Performance was good on this outdated equipment, with a small number of concurrent password changes. Resource usage at scale with a large number of users concurrently setting passwords was not measured, but anecdotal evidence suggests hardware/resource costs are minimal. Code maintenance does not incur the cost of specialized programmers because all Password Dog code is written in Python.

# 8. User Interface Developed

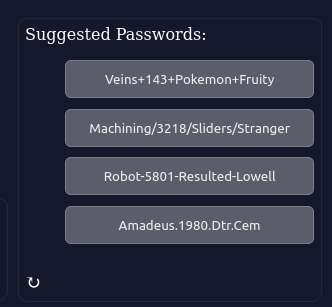
A web-based user interface was developed as a demonstration of how Password Dog might be implemented into a password change workflow. The demonstrator UI is implemented with the HTML5 “meter” element to provide a password strength meter for the user. This meter updates as the user types. Other user interfaces could be used via the JSON API provided by Password Dog.



# The source code for Password Dog is available at https://github.com/MorrisR2/password-dog

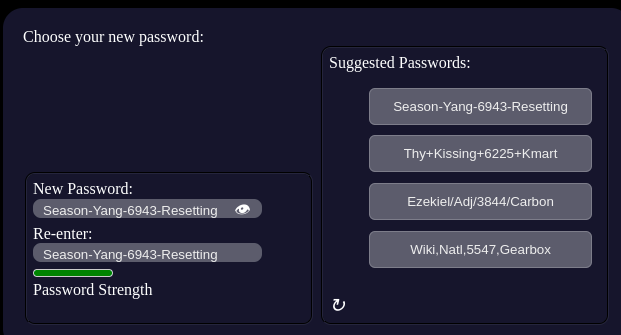
# 9. Password Generator Developed

A strong password generator was also implemented to assist users in choosing strong passwords. This generator produces passwords by selecting a number of randomly generated words from a large list as shown in this figure.



In the default configuration, the password generator selects three words from a list of 30,000. It randomly chooses one of five delimiters. For compatibility with password policies based on external standards, it capitalizes the words and also places a group of digits at a random location in the generated passwords. This algorithm provides a keyspace of 5.4×10¹⁸ with the default settings, which is adequate for most purposes. At 1 million hashes per second (DCC2), exhausting half of the keyspace would take 2.7x1012 seconds, or 85,617 years. Configuring it for four words provides a keyspace greater than 1022, for use with the fastest hashes such as MD5 (2,008 years at 65 billion per second).

The two UI components can be combined as shown for a complete UI solution.



4,000 MD5 hashes of passwords generated using this tool were submitted to Hashmob. As would be expected from the keyspace calculation, none were cracked. Discussions with members of Team Hashcat indicate that they have no method to reasonably attack passwords generated with this tool.

# 10. Summary and Conclusions

The filter proposed, implemented, and tested did perform significantly better than previously available tools for this purpose. The best of the existing tools tested scored overall 13 percentage points better than random chance; Password Dog scored 41.5 points better. Unlike studies which treat NIST or other guidelines as ground truth, this study tested against actual password crackers, determining which passwords actually get cracked and which do not.

Avenues for further improvement have been identified. Deployment of a filter/strength meter using this model for important systems that are password-protected would significantly reduce the use of weak passwords, thereby reducing security incidents.

The approach has wide applicability because it is compatible with existing COTS systems, including Windows. Rather than requiring an overhaul of the authentication processes within each system, the filter can be added as a UI element in front of the password change function. An organization could use a browser extension to apply Password Dog filtering to almost **all** web applications used by their staff, without requiring that the companies running the web sites make any changes. It remains compatible with common 2FA systems used along with passwords.

# 11. Future Work

Several possibilities for future work have been identified. The results quoted above are based on testing of a "first cut" implementation, without significant tuning and optimization. It may be useful to manually analyze the passwords that were misclassified, make some adjustments based on observed misclassifications, then compare the new predictions produced. Adjustments would include the wordlists and rule sets used. To improve the overall accuracy rate, it may be useful to test several specific changes and compare the new predictions to the actual results from the password crackers. Potential improvements that could be tested would include combining this new approach with a heuristic approach such as Zxcvbn. The testing above indicates Zxcvbn allows most passwords, including weak ones, but does catch some weak passwords. It may work better to combine the two, rejecting passwords which are rejected by either filter.

Possibly the most promising avenue for future improvement would be improvements to the inverse rule attack. This is because the rule attack is the one most used by attackers. Results of the rule attack very much depend on the wordlist used and the rule set used. To reduce that dependency, a small list of rules would be applied to the candidate password, producing several variations. Each variation would then be run through the existing inverse rule check. The total score reported would be computed based on not only the results of the candidate password itself, but also on the results from the variants produced. Variants could be produced by applying best64 or NSA64 rules to the candidate password, thereby producing 64 variations of it. If any of the 64 rules tested produces significant false positives, that rule could be removed.

The combinator attack was not implemented in this version because it isn’t used as often by attackers. An inverse combinator attack could certainly be implemented. Combinator consists of concatenating each entry in the wordlist with every other entry. Thus, the wordlist "cat, dog, horse" would produce items like catcat, catdog, cathorse, dogcat, doghorse, etc. This could be inverted by simply checking to see whether any wordlist item is the initial substring of the candidate password. If so, the algorithm would then perform the same check on the remaining portion of the candidate.

The hybrid attack is only partially implemented in this version. This is because the hybrid attack mode per se is not often used by attackers; rather, a rule attack is used to produce precisely the same result. A hybrid attack consists of concatenating items from a wordlist, like a dictionary attack, with strings produced by a mask attack. As an example, the implemented category of hybrid appends or prepends digits to the items in the wordlist. A hybrid could append or prepend any pattern of characters. This can be inverted by checking to see if an item in the wordlist is an initial or terminal substring of the candidate password. After identifying any such cases, run the existing inverted mask attack on the remainder of the candidate password.

**ACKNOWLEDGMENTS**

Royce Williams (@TychoTithonus) of Team Hashcat provided invaluable advice from the perspective of a highly skilled attacker along with performing the first round of testing, cracking test hashes.

Sam “Chick3nman” Croley, a member of Team Hashcat and Hashcat Core Developer, also provided invaluable guidance. Mr. Croley also assisted with producing part of the “ground truth” data, cracking the test hashes to determine which ones would be cracked by a very skilled attacker.

The author would like to thank the members of Hashmob for their advice and assistance. In particular, they carried out attacks on the test hashes, serving as the representative skilled attackers. Members have elected not to be identified by individual names.

Thank you to Dr. Mustaque Ahamad and Karl Grindal, instructors for the Georgia Institute of Technology practicum course. Dr. Ahamad early on steered the project in a more academic direction with careful testing and measurement. He also assisted answering questions and providing guidance along the way. Karl Grindal provided guidance on projects for the course, including this one.

Thanks also to the members of my cohort, group 4, who provided feedback at several points during the course of this work. In particular the questions they asked highlighted items that needed to be illuminated and explained in this work. Bao Nguyen, Brook Lampe, Clayton Holcomb, Fuhua Song, Jason Hodges, Matthew James, Rinat Kirimov, Somtochukwu Ochuba, and Svavar Hermannsson were very helpful.

Thank you to Kanwal Ameen for proofreading and assistance editing for formatting and clarity.

**REFERENCES**

Berger, Miriam (2020) “Man really did hack Trump’s Twitter account by guessing password, ‘maga2020!,’ Dutch prosecutors say” https://www.washingtonpost.com/world/2020/12/17/dutch-trump-twitter-password-hack/

Carolina (2018) “Northern Irish Parliament Hit by Brute Force Attack” https://www.hackread.com/northern-irish-parliament-hit-by-brute-force-attack/

Croley, Sam (aka Chick3nman], “RTX\_3090\_v6.1.1.Benchmark” <https://gist.github.com/Chick3nman/e4fcee00cb6d82874dace72106d73fef>

Epixoip “8x Nvidia GTX 1080 Hashcat Benchmarks” https://gist.github.com/epixoip/a83d38f412b4737e99bbef804a270c40 (accessed on 21 April 2022)

Hunt, Troy “Pwned Passwords” <https://haveibeenpwned.com/Passwords> (accessed on 22 April 2022)

IntelligenceCareers.gov “IC Candidate Portal” (2022) https://apply.intelligencecareers.gov/home/(home:login)

IBM ”Cost of a Data Breach Report” https://www.ibm.com/security/data-breach (accessed on 21 April 2022)

Jacquinot “History of Smart Cards” https://cardwerk.com/smart-card-history/

Kacherginsky, Peter [aka iphelix] (2019) “Password Analysis and Cracking Kit” <https://github.com/iphelix/pack>

Kamp, Poul-Henning, et al. "LinkedIn password leak: salt their hide." ACM Queue 10.6 (2012): 20. <https://queue.acm.org/detail.cfm?id=2254400&ref=fullrss>

Konoth, Radhesh Krishnan, Victor van der Veen, and Herbert Bos. "How anywhere computing just killed your phone-based two-factor authentication." International conference on financial cryptography and data security. Springer, Berlin, Heidelberg, 2016.

Odogwu, Chris (2021) “5 Times Brute Force Attacks Lead to Huge Security Breaches” https://www.makeuseof.com/brute-force/

Paganini, Pierluigi (2021 March 21) “Intern caused ‘solarwinds123’ password leak, former SolarWinds CEO says” Security Affairs h[ttps://securityaffairs.co/wordpress/115134/security/solarwinds-intern-solarwinds123-password-leak.html](https://securityaffairs.co/wordpress/115134/security/solarwinds-intern-solarwinds123-password-leak.html)

Solar Designer. (2021) “passwdqc/README ” https://www.openwall.com/passwdqc/README

Specops (2021) “Weak Windows passwords blamed for recent rise in Purple Fox attacks” https://specopssoft.com/blog/windows-passwords-blamed-for-recent-purple-fox-attacks/

Team Hashcat (2021) “Team hashcat event writeups and tools” https://github.com/hashcat/team-hashcat

Trautman, Lawrence J., and Peter C. Ormerod. "Corporate directors' and officers' cybersecurity standard of care: The Yahoo data breach." Am. UL Rev. 66 (2016): 1231.

Verizon “Data Breach Investigations Report” 2017 https://www.knowbe4.com/hubfs/rp\_DBIR\_2017\_Report\_execsummary\_en\_xg.pdf (accessed on 21 April 2022)

Wheeler, D. L. (2016). zxcvbn: Low-Budget Password Strength Estimation. 25th USENIX Security Symposium (USENIX Security 16), 157–173. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler