Shawn Jauhal

Professor Madachy

CS 558

May 16, 2019

A Look into Password Diversity

Introduction:

As technology grows to be more efficient, there becomes a problem that passwords have become less and less secure. As it becomes easier for passwords to be cracked, users must set up passwords that are more secure. A password that might seem strong at first, could take a long time for a household PC to crack, but with modern supercomputers and botnets we must think twice about creating secure passwords. A good majority of computer users try to create passwords that are memorable. Unfortunately this means password security becomes second, to creating a memorable password users would not forget.

There is an old saying that states, why build a 15,000 dollar fence for a 5,000 dollar horse. This saying applies to passwords just the same. As with any issue in cybersecurity, one of the first things to be considered is how valuable is the data that you are protecting. A large corporation can have an immense amount of security, and weak passwords can be a fatal flaw in the most secure system. Especially when corporations are considered, as the value of data rise, the importance of password security rises.

To implement this as a model we will use a model developed by National Taiwan University. They have developed a method for calculating password strength based on Weir’s algorithm. Their method of judging password strength is based after developing password patterns and pattern classes. These are defined later in this study.

The simulation would be built on a password list defined in the next section. This would allow a continuous simulation to be built based on the data provided. It would compare what percentage of the passwords have been compromised from different structures. The general method is comparing the password’s structure with other similar passwords.

To further elaborate password structure, it is the way a password is formatted. A password may contain upper-case letters, lower-case letters, digits, and symbols. These are referenced as U, L, N and O respectively. For example we could look at an example password, GoAztecs!, would have the structure, U1L1U1L5O1.

Beyond looking at how to structure a password properly, the main hypothesis for this experiment is that passwords get stronger when special characters are used. This is compared with password with strictly alphanumeric characters.

Stakeholders:

Naturally everyone uses passwords, thus anyone can be a stakeholder. Beyond that, a company that wishes to define new criteria to create passwords is also a stakeholder. To further this I oversee the training of new members of the Cyber Defense Team, and in our competition, we need to develop strong passwords. During our competitions we are in a rush for time, so creating many strong passwords is a priority.

Data Collection:

The source of data collected for this simulation is the RockYou password list. This is used frequently for many studies for password strength. It is the default password list that comes stock within Kali Linux. This is the main password list used for password cracking, thus naturally it only makes sense that it would be a proper data set for this simulation.

The RockYou password list contains over 14 million passwords, and 1,573,565 passwords that are over 12 characters. This provides the perfect list for such a simulation. It contains a real dataset, with real passwords. Thus, we are looking at real passwords, from real users.

Data Limitations:

The primary limitation of this data is that when we get more picky with our criteria for how we determine our sets of data there are smaller datasets. We our using only passwords of length 12. Running the simple python code:

**def** no\_space(password):  
 *"""  
 Checks to see if there is no whitespace within password* **:param** *password: Password being checked* **:return***: Returns true if there is no whitespace  
 """* **for** character **in** password:  
 **if** (ord(character) == 32):  
 **return False  
 return True***#Opens password list***with** open(**r"pass1.txt"**, encoding=**"utf-8"**) **as** passwords:  
 contents = passwords.readlines()  
  
password\_dictionary = []  
*#Filters password by size and makes sure there is no space***for** lines **in** contents:  
 **if** (len(lines) > 12 **and** no\_space(lines)):  
 password\_dictionary.append(lines.rstrip(**'\n'**))  
  
patterns\_list = []  
class\_list = []  
  
*#Creates class and pattern list for all passwords***for** password **in** password\_dictionary:  
 pattern\_class = **""** structure = **""** previous\_char = **''** *#Determines Pattern Class* **for** character **in** password:  
 **if** (character.islower()):  
 structure += **"L"  
 elif** (character.isupper()):  
 structure += **"U"  
 elif** (character.isdigit()):  
 structure += **"D"  
 else**:  
 structure += **"O"** patterns\_list.append(structure)  
  
 *#Determines Password Pattern* **for** character **in** password:  
 **if** (character.islower() **and** previous\_char != **'L'**):  
 pattern\_class += **"L"** previous\_char = **'L'  
 elif** (character.isupper() **and** previous\_char != **'U'**):  
 pattern\_class += **"U"** previous\_char = **'U'  
 elif** (character.isdigit() **and** previous\_char != **'D'**):  
 pattern\_class += **"D"** previous\_char = **'D'  
 elif** ((ord(character) < 48 **or** (ord(character) > 58 **and** ord(character)  
 < 65) **or** (ord(character) > 90 **and** ord(character) < 97) **or** ord(character) > 122) **and** previous\_char != **'O'**):  
 pattern\_class += **"O"** previous\_char = **'O'** class\_list.append(pattern\_class)  
  
*#Creates the dictionaries based on the lists*pattern\_dict = Counter(patterns\_list)  
class\_dict = Counter(class\_list)  
  
print(pattern\_dict)  
print(class\_dict)  
  
*#Saves dictionaries as txt files in json format***with** open(**'passwords.txt'**, **'w'**) **as** file:  
 json.dump(password\_dictionary, file)  
  
**with** open(**'patterns.txt'**, **'w'**) **as** file:  
 json.dump(pattern\_dict, file)  
  
**with** open(**'classes.txt'**, **'w'**) **as** file:  
 json.dump(class\_dict, file)

We can determine that the number of passwords, which contain at least one special character, out our dataset is 149341. While when we use passwords that only contain upper and lowercase letters, we have 555005 passwords to choose from. Lastly, we have 670038 when considering alphanumeric passwords. This leads to the limitation that although we select the same number of passwords each time from each set, we are more likely to get the same set of passwords from the list of passwords with special characters than from the other list. Naturally this is a flaw in my dataset. Although each dataset is so large we are unlikely to get repeat passwords, we are getting a better representation of the passwords in the letter only dataset.

Model Verification:

I believe we can determine the most accurate results by seeing the results of the model with various sizes of data, and seeing if its output is similar, even with varying data sizes. Also we can see how much the data varies with the alphanumeric group within each run. If that data gives consistent results within each run, we can conclude that the simulation is giving reliable data.

Experimental Design:

The simulation starts with reading all 14 million passwords from the open source list of passwords. The goal of the experiment is to estimate the probability of a password getting cracked based on a dictionary attack. We can run the simulation based on picking sets of 2500 pseudorandom passwords from the list of passwords that fit our given criteria. We have 3 sets of passwords, one with only upper and lowercase letters, the next with only alphanumeric characters, and the last containing at least one special character. Each password must have a length of at least 12. These are defined as are special character set, letter set, and alphanumeric set.

This brings us to the next hurdle in the experiment: How do we measure password strength? We can use a method described in the article: “Password Cracking Based on Learned Patterns From Disclosed Passwords,” published in the *International Journal of Innovative Computing*, from the National University of Taiwan. This article is written by Hsien-Cheng Chou , Hung-Chang Lee , Hwan-Jeu Yu , Fei-Pei Lai, Kuo-Hsuan Huang and Chih-Wen Hsueh, at the same University. They have developed a method to estimate the probability of a password being cracked based upon Weir’s Algorithm, developed by Florida State University researcher Matt Weir. They explain how this probability can be estimated based upon the structure of the password, compared to the structure of other passwords contained within the same list.

We judge passwords by its pattern. We define password patterns, and password classes. Given a password’s pattern class, we look at the probability it has the given pattern. The equation is defined as follows:

Probability(x) = xnum / Ktot

This states that the estimated probability of a password being cracked by a dictionary attack is the number of passwords with the same password structure, divided by the number of passwords found in the list with the same password pattern, divided by the total number of passwords that share the same class of pattern. This is a loose estimation, supported by the research done by National Taiwan University.

We can define a password’s pattern class based after its structure. We look at the password’s combination of upper case(U), lower case(L), numerics(N) and special characters(O). For every password we look look at its password class we start with looking at its first character. Its first character is denoted as its type, such as L, followed by a +. We then ignore all subsequent characters, until we find a character of a different type. For instance, the password computerS, would be denoted as L+U+. And we follow this pattern, until we can define the whole string. For instance, if we have a camel case password, veryStrongPassword, it would be defined as L+U+L+U+L+. We then can iterate through all the passwords, in the current subset of passwords we are observing, and find the count for each pattern class.

We can now define a password’s pattern in a similar fashion. Using the same denotation for upper case, lower case, numerics, and special characters, we can define this structure. Now we replace the pluses, as defined before, with the following amount of characters of the same type. For instance, the password listed before, veryStrongPassword, would have the pattern L4U1L5U1L7. We now go through each of our subsets and find the count for each existing pattern.

Now we can begin to estimate probabilities. For example say we found four passwords containing the pattern class L+N+. These patterns could be dog123, sdsu7, password4, and dan484. The probabilities that a dictionary cracking attack trained off of the same dictionary set cracking these passwords could now be estimated with the following process.

1. We determine the pattern class of each password

|  |  |
| --- | --- |
| dog123 | L+N+ |
| sdsu7 | L+N+ |
| password4 | L+N+ |
| dan484 | L+N+ |

Thus the Ktot of L+N+ is 4

1. We then determine the count of each pattern.

|  |  |
| --- | --- |
| dog123 | L3N3 xnum = 2 |
| sdsu7 | L4N1 xnum = 1 |
| password4 | L8N1 xnum = 1 |
| dan484 | L3N3 xnum = 2 |

1. Now we can use our equation: Probability(x) = xnum / Ktot

|  |  |
| --- | --- |
| dog123 | Probability(x) = 2 / 4 = .5 |
| sdsu7 | Probability(x) = 1 / 4 = .25 |
| password4 | Probability(x) = 1 / 4 = .25 |
| dan484 | Probability(x) = 2 / 4 = .5 |

Our code consists of three python files, store\_passwords.py, part1.py, and part2.py. store\_passwords.py generates our dictionaries used in part1 and part2. It creates 3 dictionaries, consisting of our set of passwords we are using in our code. These are our set of passwords, and their frequency. It also creates a dictionary of each password pattern, and pattern class, including their corresponding frequency. These dictionaries are stored in JSON format text files. This allows us to run the code without analyzing our entire list each time.

part1.py takes a random set of passwords from each dataset. It takes 2500 pseudorandom passwords. We analyze each password and use the equation described earlier. It finds the pattern and pattern class for each password. It finds the number of passwords in the list with the same pattern, and the number of passwords in the list with the same pattern class. Doing this we find out what degree this pattern is unique within its pattern class.

Once this is all calculated for all three of our datasets, we can compare them. We generate histograms and calculate the mean for each dataset. This way we can visualize each run and get histograms for analysis. We can see these results in the following charts:

|  |
| --- |
| A screenshot of a cell phone  Description automatically generated |
| A screenshot of a cell phone  Description automatically generated |
| A close up of a white background  Description automatically generated |

part2.py allows us to generate more specific output. It runs the simulation from part1.py, with a few simple modifications. Instead of running the simulation one time, it runs 2500 times, with 1000 random passwords. This allows us to use the formula from earlier to accurately generate the mean for each dataset. This allows us to measure what degree of diversity is achieved from each dataset. Beyond just the mean we also can calculate the standard deviation, the mean within a confidence interval of 90%, and test our hypothesis. By running these three times we can determine if passwords are more predictable if we add special characters, compared to our alphanumeric set. Here is our output from running this:

**Sample Run 1:  
The average value for the special set is: 0.037386066249383385  
The average value for the letter set is: 0.2259578439967002  
The average value for the alphanumeric set is: 0.04579006362066563  
The standard deviation for the special set is: 0.0024140959014114804  
The standard deviation for the letter set is: 0.003971175248707351  
The standard deviation for the alphanumeric set is: 0.0022585165354294242  
Means within a confidence interval of 90%:  
Special set:  
0.037306642494226945 - 0.037465490004539824  
Letter set:  
0.22582719233101772 - 0.22608849566238265  
Alphanumeric set:  
0.04571575842665 - 0.045864368814681256  
Hypothesis Test:  
False  
  
Sample Run 2:  
The average value for the special set is: 0.03742183361303022  
The average value for the letter set is: 0.22580206290052393  
The average value for the alphanumeric set is: 0.04581514994798591  
The standard deviation for the special set is: 0.002532349807272311  
The standard deviation for the letter set is: 0.003909944460249422  
The standard deviation for the alphanumeric set is: 0.0021862162689529566  
Means within a confidence interval of 90%:  
Special set:  
0.03733851930437096 - 0.03750514792168948  
Letter set:  
0.22567342572778173 - 0.22593070007326613  
Alphanumeric set:  
0.04574322343273736 - 0.045887076463234464  
Hypothesis Test:  
False  
  
Sample Run 3:  
The average value for the special set is: 0.03742295482627217  
The average value for the letter set is: 0.2259313566186554  
The average value for the alphanumeric set is: 0.045804845466331914  
The standard deviation for the special set is: 0.0025415719069589904  
The standard deviation for the letter set is: 0.0038510691107286684  
The standard deviation for the alphanumeric set is: 0.002213419476616967  
Means within a confidence interval of 90%:  
Special set:  
0.03733933711053322 - 0.03750657254201112  
Letter set:  
0.22580465644491243 - 0.22605805679239838  
Alphanumeric set:  
0.04573202396555122 - 0.04587766696711261  
Hypothesis Test:  
False  
  
Results:  
All three sample runs determine that there is no significant difference  
between the average password diversity between the two datasets.  
"""**

Furthermore, we can build on this for user input. We can build code where users can give a list of passwords and compare their structure with the passwords from our given RockYou password list.

Results:

Originally, I expected to see more password diversity amongst the set with special characters. The results from this simulation did not support my original assumptions. We tested to see, within a 90% confidence interval, if the mean probability between the alphanumeric set and special character set were significantly different. The hypothesis test showed that this was false, there is no significant difference between the two sets. Still in almost every run there is a slight difference between the two sets. Special characters proved to be slightly less predictable. This can be seen with the histograms generated from part1.py. Still there were outliers in both those sets. Simply special characters are about as good they are used, just like numbers. Special characters can be used in a predictable fashion. When create a password, we can create a phrase with multiple words. If we simply add an exclamation point on the end or set underscores between words, we generate predictable passwords. This generated outliers within our special character set. Similarly, with the alphanumeric set we had outliers. We could generate passwords with unpredictable use of numbers. We could have a password with 123 on the end, or even simply a 1. This creates a predictable password. But if a password contains special characters or alphanumeric characters at unexpected parts of the password, it becomes more difficult to crack the hash behind the password.

In conclusion think about how we could take a predictable password like the word password and add simple modifications to make it unpredictable. We could make this an alphanumeric password by adding a 123 on the end. Thus, we have Password123. We could also add a special character on the end with an exclamation point. Thus, we have, Password123!. Although these are better than just using password we can use smart word mangling rules to develop a more unique password. For sake of argument, imagine if we turned the password into pas7Swor%d. All of a sudden, we have developed a more unique password out of such a predictable word. One that is significantly less likely to be cracked by someone comparing hashes.

We still must consider that these our rough estimations, meant to give us a bigger picture of the strength of each password. We also must keep in mind that this example is a simple example, and with our larger data set we will get more accurate data. Also it must be understood that this estimation is largely meant to show what password structures our most common within our dataset, and that when we force special characters to be used in passwords, that we create much more complicated password patterns, thus will be harder to detect within a dictionary attack. Still one must keep in mind that such guidelines are not foolproof, and obvious to guess passwords, such as password!, are still easy to crack.

To ensure this experiment gives accurate results, this simulation is meant to be ran 10 times, giving different results. This will ensure that one single run is not an anomaly, and that we can be more confident with our results. We will also limit password lengths to one, and if a passwords pattern class only appear once, we will remove it from our list.

Appendix:

Resources:

Kelley, Patrick G, et al. “Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms.” *Https://Www.archive.ece.cmu.edu/*, Carnegie Mellon, 2012, [www.archive.ece.cmu.edu/~lbauer/papers/2012/oakland2012-guessing.pdf](http://www.archive.ece.cmu.edu/~lbauer/papers/2012/oakland2012-guessing.pdf).

-This article is where I learned about Weir’s algorithm, used for estimating the number of guesses to crack a password.

Chou, Hsien-Cheng, et al. “PASSWORD CRACKING BASED ON LEARNED PATTERNS FROM DISCLOSED PASSWORDS.” *International Journal of Innovative Computing*, National Taiwan University, Feb. 2013, www.ijicic.org/ijicic-11-12068.pdf.

-This article provides more details on Weir’s algorithm, and other means of estimating password strength. It also has some good diagrams and visuals to help visualize the data.