PROPOSAL: Password-to-common-English-words memorization tool

Team: A.S. and 3 others kept anonymous on github
CSE363, Prof. Dongwan Shin, NMT, Spring 2023

Introduction

In modern day scenarios, people typically rely on password managers and third-party tools for storing their passwords and sign-in information, such as Bitwarden or GoogleChrome’s PasswordManager. Though these tools tend to be reliable, they also require direct access and their own passwords and sign in methods. These obstacles are obviated if the user can serve as their own password manager by memorizing not the random and complicated strings typical of password generators, but short sequences of common language. Compression and hashing can aid the user in this context by compressing complicated, human-unreadable, and hard to memorize passwords into easily memorizable phrases. This project uses conversion due to its ability to mitigate many of the issues inherent to password retrieval for users, all while ensuring confidentiality and integrity.

Since the world is turning ever more to online resources, passwords are mostly our first line of defense when trying to protect our accounts from threats online. With the rise of cybercrime and new exploits coming out each day, it's becoming important to create strong and complex passwords to ensure the protection of our accounts and services from online attacks.

To address this, we will try to invent a tool that converts strings of ascii characters into common English words taken out of a custom dictionary. The overall purpose of this is to re-encode seemingly meaningless passwords such as “f4ac867893cb” into formats that can be more easily memorized by human users. The chosen format will be strings of words returned to the user, such as “golden fly honks useless barbell jumps”, where we expressly considered an “adjective noun verb” sentence structure and prioritize concrete words over abstract ones to ease memorization. In languages with richer pools of symbols than English, such as Japanese or Chinese, this tool may be of increased utility for users who seek to memorize information.

Some preliminary technical details

Generally, we will be working with a dictionary of common English words that has three subsections: nouns, adjectives and verbs. Each section will contain the same number of words, in the current early iteration it will be 1024 = 2^10 words, so a word count total of three times that. This means that there will be a word that will be mapped to each of the unique states that can be represented by 10 bits, and we will thus encode 10 bits out of 8 bit ascii character strings. This may, at first glance seem like not much of a gain, but the utility is that a series of words, especially in a common grammatical structure is easier to memorize than, say, 16 random characters.

Security features

We intend to generate a unique permutation of the underlying dictionary of words by shuffling it, on a per-user-basis. This means that the unique configuration of the dictionary is essential to back-calculate the original password from the series of words that were output by the tool. This likens the unique dictionary permutation to a private key, that, if undisclosed, makes the conversion from ascii byte string to word series a one-way-function (it is a one-way-function as long as the permutation of the dictionary is not known). With, say, 1024 words (we will try to make the dictionary bigger), the number of possible permutations of the dictionary is 1024!, which is computationally infeasible to brute force. The permutation will, however, be calculated based on a random seed, which means this limits the mapping, so the seed needs to be long enough to also be computationally infeasible to brute force (for example 512 bits or more may need to be used). One feature of this is that, once handed a series of words, a user may not even be aware of what password/information is encoded in it if they do not have access to the dictionary, which, in a security context, would enable them to transport sensitive information from point a to point b (in their heads) without knowing what it is. Generally speaking, another dimension of utility of this tool may be within secure password management, as a user may keep a list of word-series (on paper, in the form of mnemonics or other), but if this list is stolen, the interceptor would not have any useful information unless they also managed to steal the dictionary. The permutations of the entire dictionary - if three sub-dictionaries have a length of 1024 words each - would then be (1024!)^3, which far exceeds the number of atoms in the universe. However, a bottleneck and potential vulnerability may still be in the length of the seed used to generate the user-specific permutation, hence we need to make the seed long.

Some more detailed problem considerations

One problem that has come up is that, even if we encode 10 bits in one word, a password in bit format may not be divisible by 10, which will prompt padding and then encoding the length of the stored password in the word series output itself. We may use a special suffix word that will encode the length of the encoded password, which then makes it a proper two-way function (if the dictionary permutation is given).

Technical details and problems such as storing the permutation of the dictionary securely and where to do this, and possibly other problems, especially vulnerabilities that may emerge, are subject to future experience.

Another problem is to store information about the unique permutation in some format, in case the permuted dictionary is corrupted. This should be an option to the user, to restore the state of their dictionary. An option would be to store the seed and the hashing algorithm version that is used, however more research about the security implications of this needs to be done.

<u>Concrete example of how we expect the tool to operate</u>

Suppose a user wants to encode a very short password like “H4xx0r!!”, the algorithm would do the following:

It converts the password to bit format, this would be 01001000 00110100 01111000 01111000 00110000 01110010 00100001 00100001.

It would then re-group the bits according to the power-of-two that is currently the length of our dictionaries. As we currently have 1024 words in each sub-dictionary, we can encode at most 10 bits per word. The regrouping would then be into units of 10 bits:

0100100000 | 1101000111 | 1000011110 | 0000110000 | 0111001000 | 1000010010 | 0001 (the vertical bars are purely for ease of reading here, they will not be in the code).

The problem that emerges here is that the last 4 bits must be padded to ten, this gives:

0100100000 | 1101000111 | 1000011110 | 0000110000 | 0111001000 | 1000010010 | 0001000000

The bits are ready for encoding. The algorithm will use each of the ten byte units as an index into the (uniquely permuted) dictionary, and then append the word to the output. It will keep track of the current iteration, and apply mod 3 to it, then it will pull a word from the adjective sub-dictionary for 0, a noun for 1 and a verb for 3, so this will repeat until the encoding has finished.

Concretely, adjective_dictionary at index 0100100000 may contain the word "iridescent" and noun_dictionary[1101000111] may be "elf" and verb_dictionary[1000011110] may be "streams", then the first three words in the output will be "iridescent elf streams".

Then, because of the padding, it must keep track of the actual length of the user input, here 8, so we may use a pool of very short, reserved words as a suffix to the output to encode the length and ensure that this is a two-way function if the dictionary is known. The suffix would not be a significant vulnerability, as attackers can guess password lengths anyways because, commonly, according to statista, "Approximately six out of ten Americans had passwords between eight and 11 characters long in 2021. However, another 20 percent of respondents only created passwords with over 12 characters." So 80% of passwords are within 11 characters length.

Expected Results

We expect to deliver a desktop tool that takes an input of up to, say 32 characters, and outputs a word series that can be used to get the clear text password back. As this is a memorization tool to be used by humans, another cosmetic idea we had was to use the word series in the output as a prompt for an image generating GAN, to possibly enrich the information and make the mnemonic task easier for the user. This is optional. Apart from the memorization aspect, the tool may be able to be used in certain safe-storage, niche applications.