Making Secure Easy-to-Remember Passwords

Philip Braunstein

December 14, 2015

Abstract

Password leaks are a notorious way that systems get compromised. Passwords are often leaked exactly because they are heard to remember, which prompts people to record them in insecure locations and avoid changing them. This report describes and evaluates three password generation strategies. Three password generation strategies are evaluated in this report: random strings of characters, numbers, and symbols; abbreviations of phrases mixed with meaningful numbers; and an adjective-noun-verb-adjective-noun string modeled after xkcd 936 SOURCE: XKCD . Passwords are evaluated on ease of memorization, cryptographic strength against a password cracker, and bits of entropy

Introduction

People like to imagine that computer security is usually compromised by genius hackers exploiting inscrutable vulnerabilities. Often however, an attacker compromises a system because of seemingly stupid reasons like the being written on a note next to the computer. Shoring up technical security is a worthy cause, but this effort is rendered irrelevant unless the impact of social engineering resulting in password leaks is minimized.

People avoid changing their passwords, leave them written in plain text in a document on their computer or even on a sticky note on their computer for one reason only: secure passwords are hard to remember. Insecure and poorly-stored passwords are the cause of many security leaks FIND SOME GENERAL SOURCES . Passwords are a common source of vulnerability because different websites have different requirements for what they consider valid passwords, and passwords that are considered secure are obtuse and hard for people to remember.

In this report, three methods for making passwords are described and evaluated. The security and how easy each type of password to remember is evaluated and described in the Results section.

To the Community

Methods

Overall Description of Procedure

Each of the password generation methods described below were tested on each subject. In order to prevent bias based on subjects getting used to the memorization task, the script `determineOrder.py` was used to dictate the order that each of the password methods was tested.

For each password method, participants chose the password they wanted to use, and immediately wrote it out on paper three times in a row. After this, participants were given 1 minute to remember their password in any way of their choosing. Then all written representations were removed from the vicinity of the participant, and the participant was instructed to wait 2 minutes. After this buffer minute, participants wrote down their password to the best of their ability. For a total of 3 replicates per person, the 1 minute window to memorize the password and the 2 minute buffer window without the password were repeated two more times. After each guess of the password, the participant was permitted to see the correct password. The replicates were performed to see how well participants adapt to learning the password.

The entire procedure was explained to the participant without hiding anything as it was deemed that knowledge of the goals of the experiment would not damage the outcomes of the experiment.

Password Generation Schemes

Random String

A random string password consists of ten random characters, numbers, and symbols. A random string passwords must have contain at least one of three of the four categories: upper-case letter, lower-case letter, number, or symbol. The script `randomPass.py` generates ten of these random strings passwords. Participants selected one of the ten passwords to use.

Memorable Phrase and Number

Participants chose a memorable phrase and made an acronym of the first letter of each word. Every initial was lower-case in the password. They also chose a particular number and incorporated it after the initials of the memorable phrase. For example, I might choose the hook from a famous Rolling Stones song (**I** **c**an't **g**et **n**o **s**atisfaction, and the month and year of my graduation from college (May, 2012) to generate the following password: icgns0512.

Modified XKCD

Randall Munroe of XKCD suggests using several common words that can be used to create a more coherent memory as opposed to other password

generation methods. I have improved on his idea by forcing passwords to obey the form adjective noun verb adjective noun, where the words from these parts of speech are drawn from publicly available word lists CITE WORD LISTS .

The script `humanPass.py` creates ten of these modified XKCD passwords, from which the participant chose one. The passwords were written with all lower case letters and no spaces between the words. Participants were allowed to alter the password to fix any grammatical mistakes. For example one password generated by `humanPass.py` was `availablenervebitemolecularradish` (available nerve bite molecular radish). Participants were instructed to modify the passwords so that they make grammatical sense, but the exact modification was left to the discretion of the participant. For example, the previous password could be modified in one of two ways `availablenervebitesmolecularradish` (available nerve bites molecular radish) or `availablenervesbitemolecularradish` (available nerves bite molecular radish). Participants were encouraged to make whatever grammatical modification makes the most sense to them.

Password Evaluation

The passwords were evaluated on three criteria: the success of a participant remembering the password, the strength of the password against password cracking, and the number of bits of entropy.

Success Remembering Password

At first glance, the success of a participant remembering a password should be measured by the percent similarity of the remembered password and the actual password. This could be calculated by comparing each character in the two passwords to see how similar they are. This method, however, is insufficient. Consider the following example:

```
0123456789
023456789
```

These two passwords result in a similarity of 0.1 since the '1' is omitted in the second password. However, nine of ten of the characters were successfully remembered. Therefore, this simple metric of similarity underrepresents how similar the two passwords actually are. In short, a simple percent similarity metric does not account for character insertions or deletions in the sequences.

To compensate for this problem, the script `smartFracSim.py` calculates the fraction similar of the two passwords from the forward direction *and* from the reverse direction and averages these values (hereafter: smart fraction similar). This raises the fraction similar of the example to 0.45 $((0.1 + 0.8/2.0))$.

While the smart fraction similar is a better metric than fraction similar of only one direction, I still believed that this metric was underrepresenting the similarity in the passwords. Consider the modified xkcd

example password from above:

`availablenervebitesmolecularradish` (available nerve bites molecular radish)

Consider what would happen if a participant swapped the adjectives to make the following password:

`molecularnervebitesavailableradish` (molecular nerve bites available radish).

The original and adjective-swapped passwords have a smart fraction similar of 0.47. This significantly under-represents the how close the participant was to correctly remembering the password. The participant simply flipped the adjectives, and might score a 1.0 similarity if asked again. In order to compensate for this, the modified XKCD passwords were also analyzed for similarity as follows: what fraction of the original words were also present in the remembered password, regardless of the grammatical form (for example, `nerve` would count when matched with `nerves`). This word similarity metric was averaged with the smart fraction similar metric to compute the final similarity for the modified XKCD passwords. In the swapped adjectives example, the word fraction similar is 1.0 (all words are present). Averaged with the smart fraction similar, the final similarity fraction for this example is 0.74. The word fraction was determined by hand (by me), since it would have been difficult to write a script to recognize the varying valid grammatical forms of word as well as where to break the password string into a word.

The word fraction similarity metric is successful because it accounts for distinct parts of the password that a participant has successfully remembered. There is no exact corresponding metric for the random string and memorable phrase and number passwords. So, the fraction of characters similar between the original password and remembered password was determined for these styles of password using the script `charFracSim.py`. As above, this metric is averaged with the smart fraction similar to calculate the final similarity for random string and memorable phrase and number passwords.

Strength Against Password Cracking

Bits of Entropy

Results

Applications

Conclusion