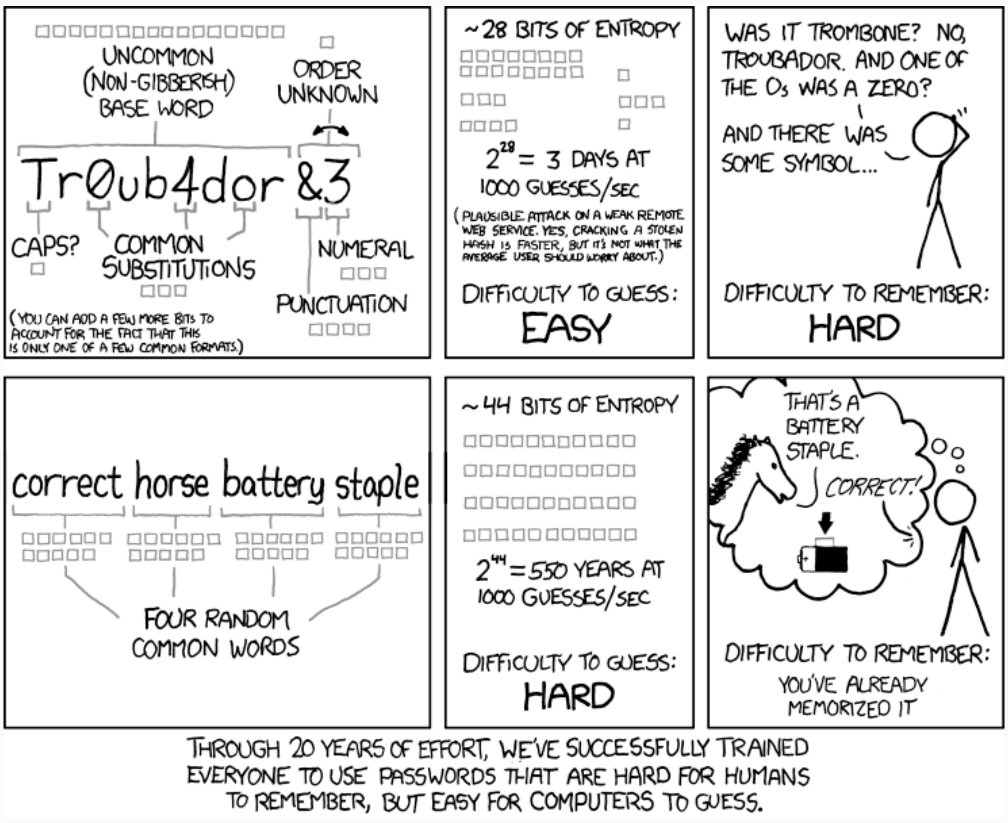
Password Generating Algorithm (Assignment #1)

Shaurya Singh

September 12, 2021

Contents



UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor &3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^28 = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS: EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER: HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^44 = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

1 Entropy and Creating Passwords

1.1 What is Entropy?

Entropy is a measure of "uncertainty" in an outcome. In this context, it can be thought of as a value representing how unpredictable the next character of of a password is. It is calculated as $\log_2(a^b)$ where a is the number of allowed symbols and b is its length.

1.2 Why `Tr0ub4dor&3` is a Bad Password

A truly random string of length 11 (not like "Tr0ub4dor&3", but more like "J4I/tyJ&Acy") has $\log_2(94^{11}) = 72.1$ bits, with 94 being the total number of letters, numbers, and symbols one can choose. However the comic shows that "Tr0ub4dor&3" has only 28 bits of entropy. This is because the password follows a simple pattern of a dictionary word + a couple extra numbers or symbols, hence the entropy calculation is more appropriately expressed with $\log_2(65000 \times$

94×94) with 65000 representing a rough estimate of all dictionary words people are likely to choose.

1.3 Creating High-Entropy Passwords You Can Remember

Another way of selecting a password is to have 2048 "symbols" (common words) and select only 4 of those symbols. $\log_2(2048^4) = 44$ bits, much better than 28.

It is absolutely true that people make passwords hard to remember because they think they are "safer", and it is certainly true that length, all other things being equal, tends to make for very strong passwords So, Instead of creating a randomly generated string of letters (which is what a random password generator would do), it makes sense to link together common word. This offers similar entropy levels, while being much easier to remember

2 But Schneier Says that the XKCD method doesn't work?

Althoguh Schneier is indeed a security genius, there are several mistakes that he makes in his analysis

1. In this case, it has always been assumed that the password generation method is known to the attacker. That's the whole point of entropy computations; see the analysis. That attackers are "on to this trick" changes nothing at all (when an attacker knows the password generation method, the entropy computation describes exactly the password strength; when the attacker is incompetent and does not know the password generation method, the password strength is only higher, by an amount which is nigh impossible to quantify).

   The quip about "passwords in memory" is just more incoherent ramblings. Passwords necessarily go to RAM at some point, whether you type them or copy-paste them from a password safe, or anything similar.

2. Picking 4 random words from a 2048-word dictionary gives you 2048*2048*2048*2048 possibilities. That's about 17.6 trillion unique phrases an attacker would have to search through. This 4-common-word-prhase has the equivalent entropy as a totally random 10-character lower case password (26 characters), 8 character upper-and-lower password with digits (52 characters), or a 7-character password with a completely random mix of letters, numbers, and digits. These are completely random characters, not the typical Word123! kind of pattern that people usually come up with, which crackers already have search algorithms for. An 8 character password with completely random character distribution is actually considered quite strong and basically impossible to crack with today's

> technology. And this is the point: that 4-word xkcd passphrase is the same strength as the impossible to crack random character string. Don't rely on your intuition. Rely on the math

Towards the end of the article, he also mentions you should use a personally memorable sentence. He isn't explicit that it should be made up by you, and two of his sentences are quotations (and famous ones at that.) This is a terrible idea, because crackers can and will index famous quotations to use them in scanning programs. It seems that Schneier is disparaging other methods, only to recommend his own inferior method.

The XKCD method works because it's random, human generated text can (and will) always be easily cracked

3 The Algorithm

3.1 Our algorithm is going to

1. Take an input (number of words in the password (`nwords`) + numbers of bits per word (`nbits`))
2. Read a wordlist file, which contains 45k words along with their frequency
3. Pick `nwords` words
4. Join those words together
5. Print the password
6. Calculate entropy for the password
7. Calculate/print what #bit key the password is equal to
8. Calculate/print how many years it will take to crack the password (with cpu, then with gpu)

3.2 Formulas used to calculate

Calculating entropy:

$$\text{entropy} = \text{number of bits} \times \text{number of words}$$

Calculating years need to crack code

$$\begin{aligned}\text{years} &= \text{entropy}/\text{crypts per second}/\text{seconds in a day}/\text{days in a year} \\ &= \text{entropy}/\text{crypts per second}/86400/365\end{aligned}$$

3.3 Sample:

The following example runs the program, telling it to create a 5 word password. Since the wordlist we use has varied word legnths, we can't calculate entropy the conventional way. However, the author has estimated the list has 11 bits per word, so the program assumes 11 bits/word by default

```
~/o/csp/assignment1 [master] λ python3 algorithm.py 5

Your password is "strike ready thought these find".
That's equivalent to a 55-bit key.
    That password would take 1.6e+02 years to crack
    on my core 2 duo from 2009, assuming an attack on a MS-Cache hash,
    (the worst password hashing algorithm in common use)
    The most common password-hashing algorithm is md5, cracking such a
    hash would take 3.2e+05 years.
    But a modern GPU can crack about 250 times as fast,
    so that same iterated MD5 would fall in 1.3e+03 years.
```

3.4 Code

The python code used to calculate this is below

```
#!/usr/bin/env python3
# Insipred by http://xkcd.com/936/

# Import what we need
import random, itertools, os, sys

def main(argv):
    # number of words should be first input from the program
    try:
        nwords = int(argv[1])
    except IndexError:
        return usage(argv[0])

    # number of bits should be second input from the program
    try:
        nbits = int(argv[2])
    except IndexError:
        nbits = 11

    # read the wordlist
    filename = os.path.join(os.environ['HOME'], 'org', 'csp',
     ↪  'assignment1', 'wordlist')
    wordlist = read_file(filename, nbits)
    if len(wordlist) ≠ 2**nbits:
        sys.stderr.write("%r contains only %d words, not %d.\n" %
                         (filename, len(wordlist), 2**nbits))
        return 2

    # generate the password, then display it
    display_password(generate_password(nwords, wordlist), nwords, nbits)
    return 0

# Info about the usage of the program, if the user gives an incorrect
 ↪  input
def usage(argv0):
    p = sys.stderr.write
    p("Usage: %s nwords [nbits]\n" % argv0)
    p("Generates a password of nwords words, each with nbits bits\n")
    p("of entropy, choosing words from the first entries in\n")
    p("<http://canonical.org/~kragen/sw/wordlist>, which is a text
     ↪  file\n")
    p("with one word per line, preceded by its frequency, most
     ↪  frequent\n")
    p("words first.\n")
    p("\nRecommended:\n")
    p("    %s 5 12\n" % argv0)
    p("    %s 6\n" % argv0)
    return 1

# function to read the wordlist file
def read_file(filename, nbits):
    return [line.split()[1] for line in
            itertools.islice(open(filename), 2**nbits)]

# function to generate the password (random words from wordlist)
def generate_password(nwords, wordlist):
    choice = random.SystemRandom().choice
    return ' '.join(choice(wordlist) for ii in range(nwords))

# function to display info about the password
def display_password(password, nwords, nbits):
    print('Your password is "%s".' % password)

    # entropy value is equal the the number of words * the number of bits
    ↪  in each word
    entropy = nwords * nbits
    print("That's equivalent to a %d-bit key." % entropy)
    print()

    # john --test (<http://www.openwall.com/john/>) reports that it
    # can do 7303000 MD5 operations per second, but I'm pretty sure
    # that's a single-core number
    t = years(entropy, 7303000)
    print("That password would take %.2g years to crack" % t)
    print("on my core 2 duo from 2009, assuming an attack on a MS-Cache
    ↪  hash,")
    print("(the worst password hashing algorithm in common use)")
    print()

    t = years(entropy, 3539)
    print("The most common password-hashing algorithm is md5, cracking
    ↪  such a hash would take %.2g years." % t)
    print()

    # <https://en.bitcoin.it/wiki/Mining_hardware_comparison> says a
    # The same mining-hardware comparison says a Radeon 5870 card can
    # do 393.46 Mhash/s for US$350.
    print("But a modern GPU can crack about 250 times as fast,")
    print("so that same iterated MD5 would fall in %.2g years." % (t /
    ↪  250))
    print()

# function to calculate years of entropy
def years(entropy, crypts_per_second):
    # entropy divided by crypts/s for inputed hash, divided by
    ↪  seconds/day, divided by days/year
    return float(2**entropy) / crypts_per_second / 86400 / 365

if __name__ == '__main__':
    sys.exit(main(sys.argv))
```