Project 3: Password Generator

Due Feb 28 by 11:59pm **Points** 100 **Available** until Mar 7 at 11:59pm

Notes:

You do not need to use Ubuntu VM to complete the project. However, before you submit, you should test your submission in an Ubuntu machine/VM to see if it works. If your project runs on Ubuntu 20.04/22.04, our autograder should also be able to run it.

Description and Deliverables

People choose weak passwords that are easily cracked because they have been taught that only confusing passwords are secure. People either reject this advice and leave themselves vulnerable or adopt password creation heuristics that are not resilient to cracking in practice (e.g., English word plus one capital letter, one random number, and one random symbol).

In this project, you will gain hands-on experience creating secure, memorable passwords resistant to cracking. To accomplish this, you will write a program that generates secure, memorable passwords using the **XKCD method** (https://www.xkcd.com/936/).

To receive full credit for this project, you will turn in the following:

1. You will write a program called *xkcdpwgen* that can generate secure, memorable passwords using the **XKCD method** (https://www.xkcd.com/936/).

This deliverable is described in greater detail below.

Program Specification

Your program may be written in any language available on the Khoury College Linux machines (this includes C, C++, Python 2 and 3, Java, Racket, Ruby, Perl, Go, Rust, and possibly others). Regardless of which language you choose, **your program must exactly obey the following command line syntax**:

```
$ ./xkcdpwgen -h
usage: xkcdpwgen [-h] [-w WORDS] [-c CAPS] [-n NUMBERS] [-s SYMBOLS]

Generate a secure, memorable password using the XKCD method

optional arguments:
    -h, --help            show this help message and exit
    -w WORDS, --words WORDS
                          include WORDS words in the password (default=4)
    -c CAPS, --caps CAPS  capitalize the first letter of CAPS random words
                          (default=0)
    -n NUMBERS, --numbers NUMBERS
                          insert NUMBERS random numbers in the password
```

```
                        (default=0)
  -s SYMBOLS, --symbols SYMBOLS
                        insert SYMBOLS random symbols in the password
                        (default=0)
```

Note that your program does not need to print this exact help text. However:

- Your program must support all five of these command-line arguments. Note that each argument is **optional**, i.e., the user may or may not supply it on the command line. Additionally, these arguments may be presented in any order on the command line.
- Your program must be named *xkcdpwgen*.

Usage of *xkcdpwgen*

By default, if you run *xkcdpwgen* with no arguments, it should produce a password composed of **four** random English words, all characters in lowercase, without numbers or symbols, like this:

```
$ ./xkcdpwgen
guacamoleexamgallopedcrediting

$ ./xkcdpwgen
flockdolliescitizenrysource

$ ./xkcdpwgen
autumnsbooboomultipliesbandwagons
```

You can use any English wordlist you wish as part of this project. Some reasonable wordlists are available **here (https://github.com/dwyl/english-words)**, **here (http://www.mieliestronk.com/wordlist.html)**, and **here (https://www.wordfrequency.info/samples_100k.asp)**. **Make sure to turn in a copy of your wordlist with your project!** (The file **must** be a line-delimited plaintext file [ie. every line contains one word], and **must** be named `words.txt`). You may assume that your program will be invoked from the same directory that contains your wordlist, and you will need to hard-code the filename of your wordlist in your program.

The "-w" and "--words" arguments allow the user to override the number of words in the generated password. For example:

```
./xkcdpwgen -w 2
studiesexaminer

$ ./xkcdpwgen -w 2
luridlypiers
```

The "-c" and "--caps" arguments capitalize the first letters of random words from the password. For example:

```
./xkcdpwgen -c 2
GrenadehostelriesBirdcagedirectives

$ ./xkcdpwgen -c 2
warehousedfootbathJiffyGazebo
```

The "-n" and "--numbers" arguments add random numerical characters into the password, either at the beginning, end, or in-between words. The "-s" and "--symbols" arguments do the same thing but for symbol characters (~!@#$%^&*.:;). For example:

```
$ ./xkcdpwgen -n 2 -s 2
@$3genteelpredatorcrickets9frustrates

$ ./xkcdpwgen -n 2 -s 4
^saltiness77checkersvulgarly$saturn^;

$ ./xkcdpwgen -n 2 -s 4
~pushes%barre^5pricksgosh$9

$ ./xkcdpwgen -n 2 -s 4
putrefying$~7polycyclic.enneads1unamended!
```

You may add additional functionality to your program if you wish, but these arguments must be available and behave exactly as specified in this project description. You may handle errors however you see fit. For example, the following invocation has an error; you may choose to display an error message or generate a "best-effort" password.

```
$ ./xkcdpwgen -c 10
```

Packaging Your Submission

Because you are allowed to program in whatever language you wish, we **require that all students submit a Makefile.** If you choose to use a compiled language, you must turn in your source code, and the Makefile must compile your program. For example, if you write your program in C/C++, the final product of the Makefile should be a program called *xkcdpwgen*.

Suppose you choose to program in a compiled language that does not produce executable binaries (e.g., the Java compiler produces .class files). In that case, you must include a shell script with your submission named *xkcdpwgen* that can (1) invoke your program and (2) forward any given command-line arguments to your program. You must also include a Makefile that transforms your source code into compiled files (e.g. .java files into .class files).

If you choose to use a language that does not need compilation (e.g., Python, Perl), **you may leave your Makefile blank.** We encourage students that choose to program in scripting languages to adopt **shebang syntax** (https://en.wikipedia.org/wiki/Shebang_(Unix)) and submit an executable script named *xkcdpwgen*.

Submitting Your Project

The exact files that you submit for this assignment will vary depending on the programming language you choose. At a minimum, you will probably submit the following:

- A *Makefile*, which may be empty
- The source code for your password generation program (named correctly)

- A `words.txt` wordlist file that is used by your password-generation program

To submit your project, do the following:

1. Create a directory *~/cy2550/project3* in the folder corresponding to your git repository.
2. Copy your Makefile, wordlist, and other source code and scripts to the *~/cy2550/project3* folder.
3. Add these files to your repository, commit them, and push the committed files to GitHub.
4. Submit your repository to Gradescope.

Grading

This project is worth 10% of your final grade, broken down as follows (out of 100):

- 20 points - turning in a password generation program that successfully compiles (if necessary) and runs on the command line, regardless of correctness
- 40 points - turning in a password generation program that has the correct default behavior, e.g., generates four-word long random passwords
- 10 points each - correct support for the words, caps, numbers, and symbols arguments

Points can be lost for turning in files in incorrect formats (e.g., not ASCII), failing to follow specified formatting or naming conventions, failing to compile, failing to follow specified command line syntax, insufficient or incorrect randomization, etc.

Tips

- How hard is the *xkcdpwgen* program to write? My reference implementation is 52 lines of Python, so not too bad.
- If you've never written a command-line-driven program before, the first step is figuring out how to read command-line arguments in your language of choice. All languages have this capability, although it's not always named the same thing. In C/C++, the command line is available as the *argc* and *argv* variables passed to *main()*. In Python, the *sys* module holds the command line arguments in the *sys.argv* variable. Take the time to look at examples of command-line parsing in your language of choice.
- If you're not sure you've implemented your program's command line correctly, have one of your friends test it out. Alternatively, post some example command lines to Piazza; we'll be happy to tell you if the formatting or behavior is incorrect.
- If you're using a compiled language, triple-check that your code compiles and that your Makefile is free of errors before you submit.
- **Always test your program on a Linux Machine before you submit it.**