Anthony Harris

Computer Security

Project Report

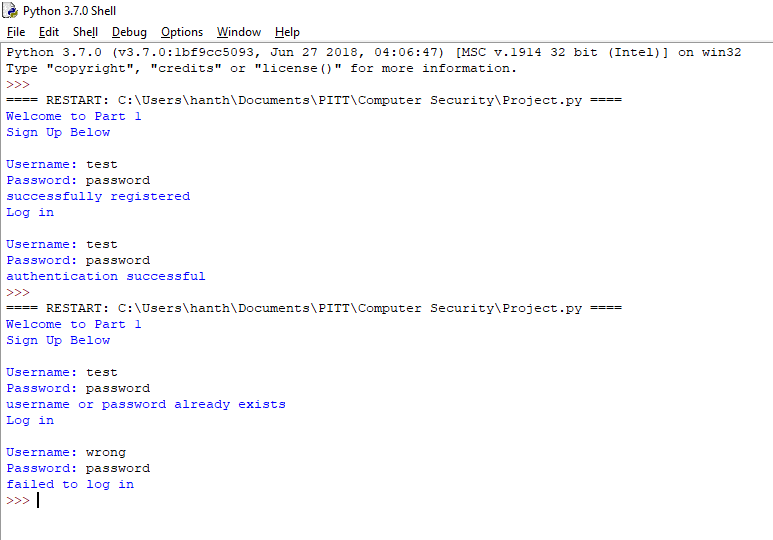
December 2, 2018

Part 1

To create a complete authentication system, a portal, storage, and verification method were required. The first step was to create a file to store login credentials, ‘login.txt’. The check (username, password) function opens the file, convert the contents to a list, hashes the password using md5, then iterates over the generated list searching for a matching combination. Finally, the function returns True or False depending on the result.

With the check () function established, the signup () function can be completed. A registering user needs to be able to check if their credentials are unique before being allowed to create an account. The signup () function prompts user input, calls the check function, and if the chosen credentials are unique, writes them to the login file.

Once theses two components are complete (storage and verification), a portal can be created to tie the authentication system together. For the purposes of this project, this portal comes in the form of a user prompt in the console. The main () function prints the start of the program, calls the signup () function, prompts the user for their credentials, checks them, and reports if authentication was successful.



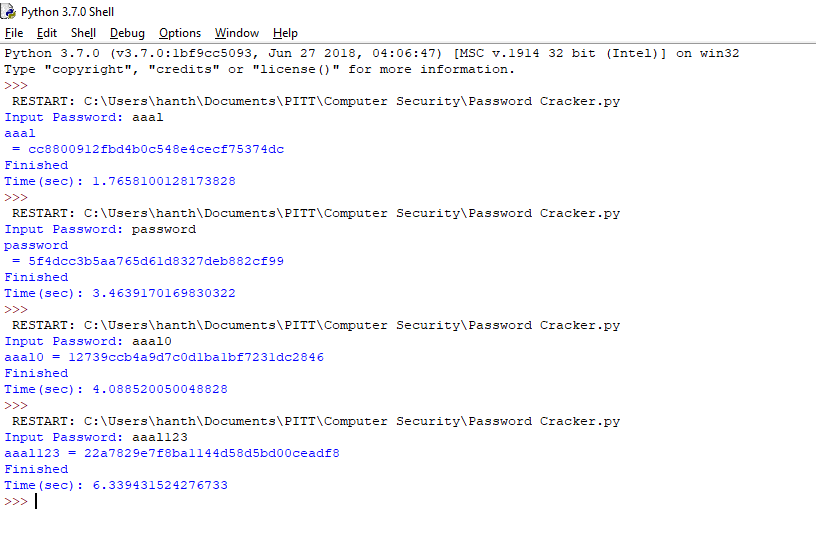
Part 2

To create a password cracker, an input must be taken and compared to a given dictionary. After initial comparison, the dictionary entries are modified into with special characters to test whether the input is a Type 2 password. If both comparisons yield no match, the attack fails. To facilitate the grader, the program accepts a normal password input, then hashes the input before passing it to the passCracker () function.

To generate all the acceptable instances of Type 2 passwords, a series of nested for loops were used to modify each entry read in from the dictionary file. Once modified, the result is hashed and compared to the hashed input of the user. If a match is found, the loop breaks (to shorten run time), and the function returns the original input or a failure message otherwise. Finally, a completion message signals the end of the program.

Alternatively, using the same series of nested for loops, a separate program was created to generate a table of all the possible Type 2 passwords and their corresponding hashes. Complimentary, the second version of the password cracker program takes in the user input, hashes it, and compares it to all the entries in the table. This method should reduce overall runtime. If a match is found it is printed with its corresponding password. Otherwise the attack fails.

Lastly the time module was imported to measure the completion time of the program. This result can vary greatly with the location (or existence) of a match.



Part 3

To create a password evaluator, a function needed to be created to take user input and compare it to a given dictionary. Weak passwords are exact matches, moderate passwords are substrings of entries, and strong passwords do not match at all. Consideration had to be taken to ensure that the function iterated through the entire dictionary before making a declaration that a password was strong. Similarly, the function should cease once an exact match is found. Because words can be substrings of other words, a weak password can be miscategorized as moderate. After evaluation, the function prints the result.

