HW #3
Security and Privacy Concepts in the Wild

Cornell Tech

Adrian Soghoian

10/28/2014

Here is a brief description of how each of the algorithms work.

Algorithm #1:
This algorithm was the most challenging to develop, but its methods are used across all three algorithms. In this case, the algorithm operates ‘blindly’ on an password without access to any password files or dictionaries.

Our original approach was to generate honeywords by both tweaking individual characters in the password (i.e. generating a similar but different string) and by tweaking *words* within the passwords (i.e. generating a similarly-constructed string but with different English-language strings). As we did not have access to a dictionary, chaffing by password proved difficult, if not impossible - we attempted to apply various grammar rules (such as counting a string as a word if it has at least one, but not all, vowels) but these rules proved unsuccessful and generally inaccurate. So, we focused on applying various small character tweaks to generate honeywords.

It made sense to us that different character types would warrant different modes of tweaking. As a user, ‘123’ and ‘password’ serve very different functions in the construction of the password ‘password123’. So, we constructed various tweaking methods for three different character types -- letters, numbers, and symbols.

Our algorithm first starts by ‘chunking’ up the string. This allows us to apply character-type-specific transformations and potentially catch instances that deserve special treatment, such as sequential numbers like “123” or birthdays represented like “19880316”. As an example of how this chunking would work, the password ‘pass123word!’ would become a Python list: [‘pass’, ‘123’, ‘word’, ‘!’]. Then, we apply specific tweaks to each ‘chunk’ of the word, such as capitalizing a random letter, shifting sequential strings of numbers, replacing one birthday with another birthday, or replacing a symbol with another. Our specific transformations are outlined in the respective files ‘letters.py’, ‘numbers.py’, and ‘symbol_split.py’, and are used throughout the rest of the homework assignment. Please note that each transformation is applied with a certain, random probability so as to prevent a pattern from being recognized and thus honeywords being identified. For each transformation, we’ll generate a large number of transformed substrings. In the reconstruction method we apply another layer of randomness in selecting which of these substrings gets chosen.

In the reconstruction phase, we zip back together the various chunks (randomly choosing among each), shuffle them, and then randomly sample an N-1 sized subset, add back in the original password, shuffle them again, and finally print them to the console.

Our algorithm receives input in the form of command-line arguments, so to generate N amount of passwords (including the honey word) execute the program like so: “A.py <password> <N>”.

Algorithm #2:

For this part of the homework, we leverage the RockYou dataset to inject another layer of falsehood into our honeyword generation by leveraging real passwords.

Our algorithm relies on the same character chaffing tricks as in #1, but supplements them by grabbing several (3) random passwords from the RockYou set and feeding them into all the logic in #1 to them as well. A fourth password is fed into the algorithm and is a ‘tough nut’. By introducing a tough nut, we hoped to create a broader spectrum of entropy across all passwords. For each of the five passwords (the original, 3 from RockYou, and 1 tough nut), we generate several character-tweaked versions and combine them all together before outputting them to the console.

Our hopes here were that interspersing variations on other passwords would minimize the patterns that could be detected when simply character chaffing one single password to generate all N honeywords. Again, probabilities are used to decide the degree to which individual chunks of characters would be tweaked.

Algorithm #3

We supplemented our approach further by leveraging even more of the RockYou dataset. Instead of randomly selecting several passwords from the top 100 to tweak (alongside with the real password and a tough nut), we decided to calculate the Levenshtein Distance from the original password to every entry in the RockYou dataset, and then sorted the values. Originally we planned to then select the top 4 most similar passwords to tweak, but results were not optimal. For example, the test case of ‘password’ generated variations like ‘Password’, ‘password1’, and ‘password3’. We felt that this direction created an opportunity for an attacker to (quite reasonably) look at these slight variations and select the one with the lowest entropy.

So, instead of selecting the top 4 most similar passwords, we randomly selected 4 passwords from a specific subset of the RockYou list. This subset corresponded to the 2,000th most similar password through the 100,000th most similar password. This subset seems to have several advantages: it produces similarly structured passwords to the original, the passwords have a low probability of sharing a proper noun with the original password, and they represent a broad spectrum of entropy. Of course, for each of the 4 passwords we randomnly select from this subset, we apply our same modes of character chaffing, shuffling, and random selection from part 1 before returning them as honeywords to the user. Again, these tweaked passwords -- like in part 2 -- supplement the character chaffing being done directly on the *original password*.