Jennifer Fairhurst September 14, 2024

6.160 Lab 0

Collaborators: *none*

Problem 1

A) Because the adversary (other users that have accounts in the computer) will be able to pregenerate hashes for common/many passwords using the hash function and salt that's specified in the shadow file and will be able to compare them to the hashed passwords and log into all the user's accounts without running into issues of limited password attempts.

B) Because it allows for more secure hash functions to be added to the system later without forcing all passwords to be updated and it also allows for admin/root user/users to pick hash functions that best suite their security preferences (either on a per password basis or per user basis.)

C) It allows for the hashing computation to be significantly slower and it becomes more computationally expensive to use that hash function. This means that adversaries can't precompute hashes for many values/passwords (it isn't as easy to do an brute force/offline dictionary attacks.)

D) I would set d to be as high as possible such that the user won't notice it or such that it'll be an acceptible wait-time for authentication (e.g. a few seconds) but not so high that the user will be frustrated (around 8 seconds). We need it to be as high as possible because everything related to the student, including extremely sensitive information and critical functions (e.g. charging the student account, modifying classes, ...), is secured behind that authentication wall on a public domain that anyone can access and attempt to hack. In other words, we need to protect against brute force/offline dictionary attacks.

E) No, I would set d to be lower because the threat model is different since accessing my laptop isn't easy (i.e., someone would have to steal my laptop to be able to start doing any damage/attempts or I'd have to leave my laptop unattended which doesn't happen) and it isn't online. Additionally, I expect logging into my laptop to be faster.

Problem 2

B) Since we have 26^{20} potential passwords that can map to 2^{256} outputs from the cryptographic hash function, we can see that there are no collisions to consider. Let k be the number of guesses we made and we know that at most, we will have to guess $26^{20} - 1$ times (since we'll know that the only combination not tried must be correct). Therefore, the expected number of guesses to recover the password is:

$$\mathbb{E} = \sum_{k=0}^{26^{20}-1} k * \frac{1}{26^{20}} = \frac{1}{26^{20}} * \sum_{k=0}^{26^{20}-1} k = \frac{1}{26^{20}} * \frac{(26^{20}-1) * 26^{20}}{2} = \frac{(26^{20}-1)}{2}$$

D) It would increase significantly because we will need to hash every password we randomly sample with every unique salt in hashes.txt which adds a huge overhead to the speed and computational cost.

Problem 4

A) The birthday paradox specifies that given a hash function with a 56 bit output, one can always find a collision in O($\sqrt{2^{56}}$). In other words, you'll need $\sqrt{2^{56}} = 2^{28}$ attempts to find a collision and therefore, you won't need to store more than that if you have a hash table. However, to avoid memory overheads and store a lot less than 2^{28} bits, we need to take advantage of the fact that there is a cycle that repeats and that we could take varying steps sizes while moving in the graph (i.e., while hashing). So, we need two independent hashing variables/pointers going at different speeds (One hashing once (H(x)) and the other, double hashing (H(H(x)))). Eventually, these two variables/pointers will meet indicating a collision and the presence of a cycle.