COMP2700: solutions to selected exercises from Lab 4

Exercise 1 The purpose of storing passwords in the hash form is to make it more difficult for an attacker to perform an offline guessing attack. If the hashing is done at the server side, every time a user (or an attacker impersonating a user) wants to log in, he or she would have to send the actual password, which the server then hashes and compares to the hashes in the password file. So in the case where an attacker gets hold of (parts of) the hashed password file, the attacker would still need to reverse the hashes to obtain the actual passwords in order to log in to the server. Since hash functions are typically one-way function, this task will be computationally infeasible.

If, however, hashing is done at the client side, and the server only needs to compare the hash sent by the client with the hashes in the password file, then if the attacker manages to get hold of the hashed passwords, the attacker does not need to reverse the hashes to obtain the actual passwords. All the attacker has to do is send the server the hashed password of the account it wants to access.

In other words, if hashing is done at the client side, there is no point in doing the hashing at all since the hash in this case serves as a password!

Note that this authentication scheme is actually not a hypothetical scheme; it was used in Windows NTLM authentication, leading to an attack technique called pass-the-hash.

Exercise 2

1. There are 26 possible choices for each character in a password, so there are 26^n possible passwords of length n. So for $n = 5$, the number of possible passwords is 26^5.
2. Assuming the password is of length at least 1, there are $\Sigma_{i=1}^{n}(26+26)^i$ passwords of length n. So for $n = 5$ we have
$$\Sigma_{i=1}^{5} 52^i = 387659012$$
passwords.
3. For this exercise, it is easier to first think about the number of invalid passwords (passwords that don't comply with the policy). An invalid password is a password that contains no digit, i.e., each character is either a lower case letter or an upper case letter. Since the password length is 10, there are $(26+26)^{10}$ invalid passwords. The number of passwords that contain at least one digit is then obtained by subtracting invalid passwords from the set of all possible combinations of letters and digits, giving us:
$$(26+26+10)^{10} - (26+26)^{10} = 62^{10} - 52^{10} = 694744259919283200.$$

Exercise 3.

1. This is an instance of the verification problem. The probability that the selected biometric fails to match the user's biometric is given by
$$\text{FRR} = \text{FTA} + \text{FNMR} \times (1 - \text{FTA})$$
Since we assume FTA = 0%, in this case we have FRR = FNMR = 0.8%.
2. This is an instance of the identification problem. The formula to use is FPIR (False Positive Identification Rate): for a database with n biometric templates, the probability that at least one template will match a given biometric sample not already in the database is given by:
$$\text{FPIR} = (1 - \text{FTA}) \cdot (1 - (1 - \text{FMR})^n)$$
So the probability that Alice's fingerprint matches one of the stored fingerprint is
$$(1-0) \times (1 - (1 - 0.005)^{200}) = 1 - (0.995)^{200} \approx 0.633 = 63.3\%.$$
3. To succeed in creating another account, Bob's fingerprints will have to fail to match all registered fingerprints, including his own. The failure to match Bob's scanned fingerprints against another person's fingerprints in the server's database is given by $(1 - \text{FMR})$, whereas the failure to match Bob's scanned prints against his own stored fingerprints is given by FNMR. So the probability that Bob's fingerprints do not match any of the prints in the server database is given by:
$$(1 - \text{FMR})^{199} \times \text{FNMR} = (1 - 0.005)^{199} \times 0.008 \approx 0.003 = 0.3\%.$$

Exercise 4.

1. In this case you have a non-case sensitive version of a password, and you are asked to guess the case sensitive version. For each character in the non-case sensitive password, it is either capitalized in the case sensitive version or it is not. As an example, if the non-case sensitive password is ABC. Then all possible case-sensitive versions are ABC, ABc, AbC, Abc, aBC, aBc, abC, abc.

   Suppose the length of the password (case sensitive or not) is n. For each character at position i, for $1 \leq i \leq n$, in the non-case sensitive password, we need to make two guesses (whether it is capitalized or not). So in the worst case, we need to make 2^n guesses to find the correct case-sensitive version of the password.

2. The first strategy (brute force LM hash followed by guessing case-sensitivity) requires, in the worst case: $26^n + 2^n$ attempts. The second strategy requires $(26 + 26)^n = 52^n > 26^n + 2^n$ for $n \geq 1$ so the first strategy is obviously better.

Exercise 5 Example solution:

```
$ hashcat -a 0 -m 0 md5_hashes.txt  /usr/share/dict/cracklib-small -o attack1.txt --force
```

Note that the option –force is needed only if you're running hashcat in the VM. This gives you three passwords:

```
5d41402abc4b2a76b9719d911017c592:hello
5f4dcc3b5aa765d61d8327deb882cf99:password
008c5926ca861023c1d2a36653fd88e2:whatever
```

When run in the Azure Lab VM, this gives a hash rate of 186 kH/s (but your number may be different).

Exericse 6 Brute force one character up to 4 characters using character set ?a (lower case, upper case, digits, symbols).

```
$ hashcat -a 3 -m 0 md5_hashes.txt ?a -o attack2.txt --force
$ hashcat -a 3 -m 0 md5_hashes.txt ?a?a -o attack2.txt --force
$ hashcat -a 3 -m 0 md5_hashes.txt ?a?a?a -o attack2.txt --force
$ hashcat -a 3 -m 0 md5_hashes.txt ?a?a?a?a -o attack2.txt --force
```

For brute forcing 5 characters, the character set ?a might be too large so it will take much longer (estimate how much longer it would take compared to brute forcing ?a?a?a?a) to run in the lab VM. So we try a smaller character set, using only lower case letters and digits, by defining a custom character '-1 ?l?d'.

```
$ hashcat -a 3 -m 0 md5_hashes.txt -1 ?l?d ?1?1?1?1?1 -o attack2.txt --force
```

For passwords of length 6, try even smaller character set, e.g., digits (?d).

```
$ hashcat -a 3 -m 0 md5_hashes.txt ?d?d?d?d?d?d -o attack2.txt --force
```

These give additional five passwords:

```
081dc9bdb52d04dc20036dbd8313ed055:1234
b6fce64f8f5d01434b1b3feb09e42d7d:xu78
033bd9ac06bf7f5b3bac657e1dcc8dac:ghiel
033bd9ac06bf7f5b3bac657e1dcc8dac:ghiel
e10adc3949ba59abbe56e057f20f883e:123456
```

Exercise 7 Here are some rules (from the lab4.rule file):

```
# ==========================================================
c

$0
$1
$2
$3
$4
```

```
$5
$6
$7
$8
$9

# 4 digit suffix (years)

$2$0$1$8
$2$0$1$9
$2$0$2$0

# common suffix to words
$e$d
$i$n$g
$y
$e


# substitution: replace letters with similar looking numbers
si1 so0

# combination

# capitalisation + two digit suffixes
c $0$0
c $0$1
c $0$2
c $0$3
c $0$4
c $0$5
c $0$6
c $0$7
c $0$8
c $0$9

# substitutions + one digit suffixes
so0 $1

# capitalisation + substitutions + suffixes
# add more patterns as needed

c sa4 so0 $1
# ================================================================
```

To use these rules, run the following command (note: the character \(backslash) followed by a new line allows us to write a command that spans two lines in Linux terminal; but you could also just write it in one line. It is separated into two lines here to fit the layout of this document).

```
$ hashcat -a 0 -m 0 md5_hashes.txt /usr/share/dict/cracklib-small \
  -r lab4.rule -o attack3.txt --force
```

This give us a few more passwords:

```
47bfdc6e045a09a0c010e58806e955e6:Di4m0nd1
d2cd3371597c0d1ee981493545bb2895:h1ck0ry
b4c0be7d7e97ab74c13091b76825cf39:passw0rd1
b9ba865fec061c9706d2fd7ce49c0cc7:Purple
5cd13fb5013439ceac973b47da9d5bcc:sadie2020
5dca2e03b49fe7dc794ff47f6bc70d0b:Wesley07
```

So far we have covered 14 out of 16 hashes. For the remaining, we'll just use a rule set provided by hashcat (best64.rule):

```
$ hashcat -a 0 -m 0 md5_hashes.txt /usr/share/dict/cracklib-small \
  -r /usr/share/hashcat/rules/best64.rule -o attack2.txt --force
```

This gives us the remaing passwords:

```
841f7df2067aadcb88eda6137c6ec414:1robin
2951b7cff8784139bd4eb9a1703bb341:scrappy
```

Exercise 8. When run in the Azure Lab VM, the hash rate was around 208 H/s. Compared to the hash rate we get in Ex. 5, this is almost about 900 times worst!

Exercise 9. Let X be the set of all passwords. Then

$$Pr[X = p] = \frac{0.6}{10^4}$$

if p is an English word, and

$$Pr[X = p] = \frac{0.4}{26^5 - 10^4}$$

otherwise. The entropy in this case is given by:

$$\begin{aligned} H(X) &= \sum_{i=1}^{26^5} p_i \log_2(\frac{1}{p_i}) \\ &= \sum_{i=1}^{10^4} \frac{0.6}{10^4} \log_2(\frac{10^4}{0.6}) + \sum_{i=1}^{26^5-10^4} \frac{0.4}{26^5 - 10^4} \log_2(\frac{26^5 - 10^4}{0.4}) \\ &= 0.6 \log_2(\frac{10^4}{0.6}) + 0.4 \log_2(\frac{26^5 - 10^4}{0.4}) \approx 18.34. \end{aligned}$$

Exercise 10. Since FTA is 0%, we only need to take into account the error rates introduced by the algorithms of the biometric matching. Let F_1 be the FMNR of the fingerprint scanner and let F_2 be the FNMR of the iris scanner. Let us first calculate the probability that Alice *succeeds* in authenticating herself to the system; once we obtain this probability, we then subtract it from 1 to obtain the probability of Alice failing to authenticate. For Alice to succeed in authenticating herself, her fingerprint sample must match the stored reference fingerprint, *and* her sample iris scan must match her stored iris scan; this gives us the probability of Alice succeeding ps:

$$ps = (1 - F_1) \times (1 - F_2)$$

so the probability of her failing to authenticate is

$$1 - ps = 1 = (1 - F_1) \times (1 - F_2) = 1 - (1 - 0,02) \times (1 - 0.001) = 0.02098.$$