Homework 3

- Homework 3
  - Exercise 1: [attack] Password Cracking
    - Part 1a: Brute force attack
    - Part 1b: Dictionary attack with rules
    - Part 1c: Dictionary attack with salt
  - Exercise 2: [attack] Use rainbow tables
    - What's the point?
    - Who are we attacking ?
    - The characters set
    - Generate a Rainbow table
      - The reduction functions
      - Computing the table
    - Attacking the hashes
    - Is this feasible?
    - Additional "theoretical" questions

Exercise 1: [attack] Password Cracking

The objective of this exercise is for you to crack some passwords. The exercise consists of three parts. In each part, the setting is slightly different modifying the way the cracking should be executed.

For each part, please find the set of password to crack as a list of hash digests.

Part 1a: Brute force attack

In this part you should implement a brute-force attack. Passwords are randomly generated from the set of lowercase letters and digits ('abcd. . . xyz0123. . . 9') and have length 4, 5, or 6 characters. Generated passwords are then hashed with SHA-256 and corresponding hexdigests are sent to you in the file.

The list of SHA-256 digests you need to crack is:

- `7c58133ee543d78a9fce240ba7a273f37511bfe6835c04e3edf66f308e9bc6e5`
- `37a2b469df9fc4d31f35f26ddc1168fe03f2361e329d92f4f2ef04af09741fb9`
- `19dbaf86488ec08ba7a824b33571ce427e318d14fc84d3d764bd21ecb29c34ca`
- `06240d77c297bb8bd727d5538a9121039911467c8bb871a935c84a5cfe8291e4`
- `f5cd3218d18978d6e5ef95dd8c2088b7cde533c217cfef4850dd4b6fa0deef72`
- `dd9ad1f17965325e4e5de2656152e8a5fce92b1c175947b485833cde0c824d64`
- `845e7c74bc1b5532fe05a1e682b9781e273498af73f401a099d324fa99121c99`
- `a6fb7de5b5e11b29bc232c5b5cd3044ca4b70f2cf421dc02b5798a7f68fc0523`
- `1035f3e1491315d6eaf53f7e9fecf3b81e00139df2720ae361868c609815039c`
- `10dccbaff60f7c6c0217692ad978b52bf036caf81bfcd90bfc9c0552181da85a`

What can you say about the computational time required? Is this kind of attack parallelisable? Can you explain in one sentence the rationale behind the use of SHA-256?

Part 1b: Dictionary attack with rules

In the previous part, you implemented the brute force attack and faced one of its drawbacks. Unfortunately, people very rarely use random passwords. Instead they use some common words and sometimes modify them slightly. This is a fortunate fact for password crackers, because they can use ‘dictionary attacks’ to crack the passwords more efficiently than with brute-force. In this part you should implement one such dictionary attack. We generate a password by selecting a word from a large dictionary and then randomly applying some of the common user modifications:

- capitalise the first letter and every letter which comes after a digit (for example: ‘com402class’ becomes ‘Com402Class’). If you are using Python, this is easily achieved by ‘title()’ function from string module (‘com402class’.title() will give you ‘Com402Class’)
- change ‘e’ to ‘3’
- change ‘o’ to ‘0’ (that’s small letter ‘o’ to zero)
- change ‘i’ to ‘1’

Note that those operations are not all commutative. (For instance, ‘window’ can become ‘W1ndow’, or ‘w1nd0w’, or W1Ndow,. . .)

Examples of dictionaries can be found online (e.g. https://wiki.skullsecurity.org/Passwords).

The list of SHA-256 digests you need to crack is:

- `2e41f7133fd134335f566736c03cc02621a03a4d21954c3bec6a1f2807e87b8a`
- `7987d2f5f930524a31e0716314c2710c89ae849b4e51a563be67c82344bcc8da`
- `076f8c265a856303ac6ae57539140e88a3cbce2a2197b872ba6894132ccf92fb`
- `b1ea522fd21e8fe242136488428b8604b83acea430d6fcd36159973f48b1102e`
- `3992b888e772681224099302a5eeb6f8cf27530f7510f0cce1f26e79fdf8ea21`
- `326e90c0d2e7073d578976d120a4071f83ce6b7bc89c16ecb215d99b3d51a29b`
- `269398301262810bdf542150a2c1b81ffe0e1282856058a0e26bda91512cfdc4`
- `4fbee71939b9a46db36a3b0feb3d04668692fa020d30909c12b6e00c2d902c31`
- `55c5a78379afce32da9d633ffe6a7a58fa06f9bbe66ba82af61838be400d624e`
- `5106610b8ac6bc9da787a89bf577e888bce9c07e09e6caaf780d2288c3ec1f0c`

What do you observe compared to part 1a?

Part 1c: Dictionary attack with salt

In the previous part of the exercise you implemented a dictionary attack. You should notice that once you have a dictionary you can compute the hashes of all those words in it, and create a lookup table. This way, each next password you want to crack is nothing more than a query in the lookup table. To tackle this problem, passwords are usually ‘salted’ before hashing. Salt is exactly two characters long and it contains only hexadecimal characters. In this part of the exercise you should implement another attack using a dictionary. We generate a password by simply selecting a random word from a dictionary and appending a random salt to it. The password is then hashed with SHA-256 and hexdigest and salt are available to you. Your task is to crack the passwords using a dictionary.

The list of SHA-256 digests you need to crack and the salt (in brackets) used are:

- `962642e330bd50792f647c1bf71895c5990be4ebf6b3ca60332befd732aed56c` (`b9`)
- `8eef79d547f7a6d6a79329be3c7035f8e377f9e629cd9756936ec233969a45a3` (`be`)
- `e71067887d50ce854545afdd75d10fa80b841b98bb13272cf4be7ef0619c7dab` (`bc`)
- `889a22781ef9b72b7689d9982bb3e22d31b6d7cc04db7571178a4496dc5ee128` (`72`)
- `6a16f9c6d9542a55c1560c65f25540672db6b6e121a6ba91ee5745dabdc4f208` (`9f`)

- `2317603823a03507c8d7b2970229ee267d22192b8bb8760bb5fcef2cf4c09edf` (17)
- `c6c51f8a7319a7d0985babe1b6e4f5c329403d082e05e83d7b9d0bf55876ecdc` (94)
- `c01304fc36655dd37b5aa8ca96d34382ed9248b87650fffcd6ec70c9342bf451` (`7f`)
- `cff39d9be689f0fc7725a43c3bdc7f5be012c840b9db9b547e6e3c454a076fc8` (2e)
- `662ab7be194cee762494c6d725f29ef6321519035bfb15817e84342829728891` (24)

Why is it a good idea to salt the passwords? Estimate the complexity required in this part if the salts were not provided. What additional security countermeasures could you think off?

Note1: The SHA-256 digest of a password 'psswd' is the result of SHA-256(psswd). The digest of a salted password with salt XX is SHA-256(psswdXX).

Note2: Not all dictionaries are the same, be aware that if you implement the attack correctly but you can't crack the passwords, then you might be using a dictionary which doesn't contain all the words as the dictionary we used.

Note3: in order to check the passwords, just compute the hash of the word (with or without salt):

- macOS cli: `echo -n "psswd" | shasum -a 256`
- unix cli: `echo -n "psswd" | shasum -a 256`
- Windows: look into *Microsoft File Checksum Integrity Verifier* or use an online hasher

Note4: The attacks you implement might take some time when you run them. Depending on your hardware and your implementation, the attacks may run more than 30 minutes.

Exercise 2: [attack] Use rainbow tables

In this exercise, you will attack a website with weak password requirements, using a rainbow table. In the first part, you will generate a rainbow table, and store it as a file. Then, you will use it to attack some passwords.

What's the point?

Using a rainbow table is a great application of the space/time tradeoff. It's like precomputing everything, but you don't have to store it. You only store a fraction of that. If you used 100 reduction functions, you will store (roughly) only one percent of all the precomputed hashes.

This is great for attacking a lot of passwords, if you have some spare space. The more space you use (the more passwords you precompute), the faster will the following attacks be. If you table is relatively "thin" (few reduction functions), you won't save much space compared to storing everything, but each lookup will be faster. With a "fat" table (many reduction functions), you save more space but each lookup is more costly. Hence, the tradeoff.

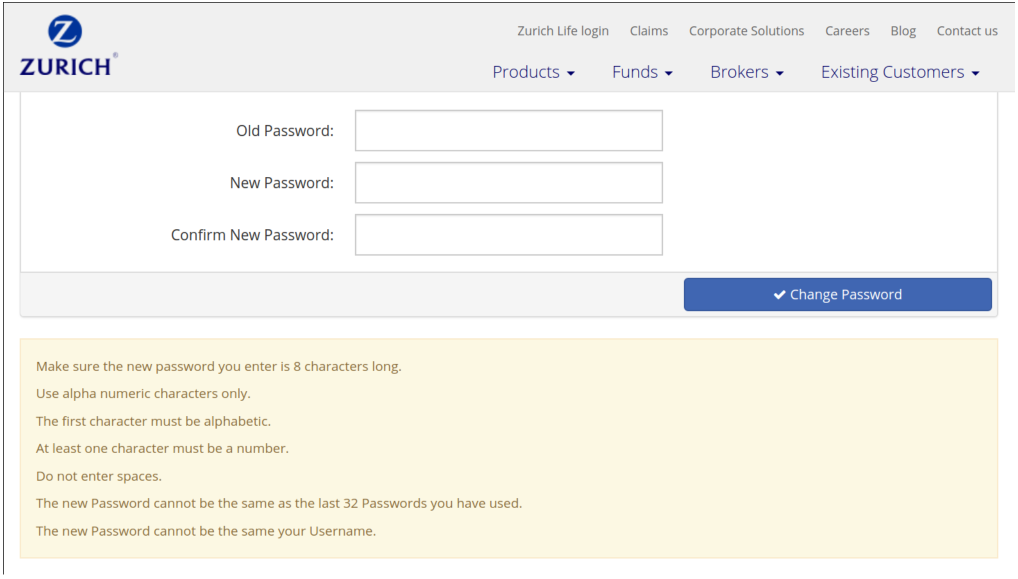
Who are we attacking ?

Suppose Zürich Insurance was to be hacked, and a dump[1] of the passwords was leaked. Because they are not completely oblivious to security, the passwords are not stored in plaintext, but hashed, using **SHA256 without salt**. You wish to reverse the hashes (and inform the users of the vulnerability, because you are a White Hat Hacker).

[1] Note: obviously, this dump is not real, and doesn't come from a leak at Zürich insurance; we generated it randomly using the rules for the characters set we define below

The characters set

As we saw earlier, trying to reverse hashes without any idea of the charset of passwords is generally a bad idea. Fortunately, Zürich Insurance has some very *very* weak requirements for passwords:



Old Password:

New Password:

Confirm New Password:

Make sure the new password you enter is 8 characters long.

Use alpha numeric characters only.

The first character must be alphabetic.

At least one character must be a number.

Do not enter spaces.

The new Password cannot be the same as the last 32 Passwords you have used.

The new Password cannot be the same your Username.

(credits to dumb password rules for the image)

Now, that's awesome (for us). The passwords we are attacking are **only alphanumeric** and with a **length of exactly 8**. We are changing two requirements here: The passwords must also be lowercase, and we relax the condition on the first character (it can also be a number). The first change is to make the hacks "more feasible", while the latter is for convenience when defining the reduction functions.

In short, the passwords set will be as follow: * Exactly 8 characters * Alphanumeric (a-z and 0-9) * Lowercase

Generate a Rainbow table

Now that you have your passwords set, you can start thinking about the rainbow table. The 3 steps are the following:

1. Define a hashing function (SHA256)
2. Define a set of reduction functions (see below)
3. Compute all rows of the table, and store the first and last columns (plaintexts).

The reduction functions There are plenty of convenient reduction functions that can be used. Before starting, try to think of one. This exercise is not as trivial as it seems. How can you go from a hexadecimal hash, to an arbitrary set of characters, of fixed length?

Arguably the most practical for us, is one that makes use of viewing a number in a different base. Let's first define *one* reduction function, and the others will follow easily.

Let's say you have a (short) hash: `a9b6f58c0`. Viewing this as an integer in base 16, you get the corresponding integer 45557438656 (base 10). Then, you consider your charset: for this example, we will use `abcdefghijkl` (that's kind of base 12); of course, a base 36 charset (a-z0-9) will have to be used for the final exercise. Express your integer in the new base, character by character, until you reach your desired length (8, for us). You get your first character by doing `45557438656 % 12 = 4`; you then take the 5th character of the charset, that is `e`. Now repeat the operation on `45557438656 // 12 = 3796453221` (an integer division). Do this until you have a string of the desired length. The above string will yield the reduction `ejfgbbfl` (for a length 8 and the 12-characters charset). Make sure you understand why and how.

But with only one reduction function, this ain't no rainbow table, it's a Hellman Table. The very power of rainbow table is the use of **multiple** reduction functions. With this method, we can easily define an arbitrary number of different functions: Once you've converted your hash to an integer, add to it the index of the column. 45557438656 will become 45557438686 in column 30.

Computing the table Each row begins with a different string. You can chose to generate random string or, even simpler, pass the row number through the reductio function. Then, you apply successively the hashing and reductions until you have the final plaintext. After that, you store the first and last plaintext (preferably in a structure like a dictionary). Once you have computed all the rows, write the table to a file (we suggest to use pickle).

It's up to you to chose the dimensions of the table.

Attacking the hashes

Now that you have an "offline" table, time to attack hashes. Wikipedia has a great explanation on this. With your hash, apply the last reduction function to it (say, f99). This gives a potential password. Look if it is in the last column of your table. If not, start over: take your hash, apply the two last reduction functions (f98, then hash, then f99), and look if it is in the last column. Repeat until you have a match.

Once you have a match, take the first plaintext of the row, and recompute the whole row, until you find your target hash (or the end of the row).

It is possible that you don't find the preimage. In this case, continue the process of adding reduction functions, until you have another match. Take a second to think why this is possible.

Take the dump and reverse as many hashes as possible. How many can you find with your table?

Is this feasible?

A quick note on feasibility here. Obviously, the passwords space we are attacking is huge (368 = 2.8 x 1012). Your table can't cover everything, it will have a certain *success rate*. The bigger the table, the higher the rate. You are not supposed to reverse all of them. We obtained a success rate of ~3.5% with a 10M rows, 1k columns table.

Additional "theoretical" questions

Now that you are an expert on Rainbow Tables, here are some additional questions. Try answering them.

1. How do you compute the success rate of your table?
2. What are the advantages of making the table "fatter" (more columns)? And "taller" (more rows)?
3. How does height and width relate the the size (on disk) of the table, and the lookup time?
4. The state-of-the-art cracking uses multiple tables. What is the point?
5. What would be the (estimated) size of a table to have a >95% success rate on this passwords space?
6. Can you parallelize the processes of creating the table and hacking?

7. How would you modify the reduction functions to match exactly the Zürich Insurance requirements? How would that affect your success rate?