Assignment 6: Database Design Patterns (Password Storage)

Due: Saturday Feb. 26th 11:30PM (you should aim to finish mid-week though; Part A really isn't a long assignment and we want you to have time to work on your Final Project/meet with El if you haven't for Part B requirements).

In this assignment you will explore an important design pattern used in relational database schema design: how to manage user authentication without storing passwords in an insecure way. This will be a shorter assignment (moved before A7's ER diagrams this term), but you will ultimately incorporate it into your Final Project (possibly with slight modifications depending on your representation of a users table). For ease of testing, you will complete the following exercises independent of your project though.

A start to **`setup-passwords.sql`** is provided with this assignment, so that you can fill in the answers between the required annotations for Part A.

Part A: Secure Password Storage (40 points)

One component of the Final Project gives you practice exploring a very important design pattern used in relational database schema design: how to manage user authentication without storing passwords in an insecure way.

Review Lecture 02/22 material for an overview motivating password hashing and salts.

Overview of Password Hashing and Salts

It is unfortunately all too common for database schemas to include a severe design defect: storing user passwords in plain text in the database. This is a catastrophic flaw for several reasons. First, once the database is compromised, the attacker can use users' passwords to access the system through normal channels. Second, most users use the same passwords for many different systems; once a user's password is known, that password can be used to compromise other systems that the user has access to. Your Final Project most likely stores some notion of user information, and this section will walk you through how to implement secure password management (fortunately, we get most of the hashing functionality available in MySQL!)

As discussed in lecture, a simple solution is to apply a cryptographically secure hash function to passwords. Instead of storing the password itself, a *hash* of the password is stored. Authentication simply involves applying the hash to the entered password, and then comparing the hash values. Unfortunately, this approach is susceptible to *dictionary attacks*; users are very predictable, and by computing the hash codes of the most common passwords (e.g. "abc123", "password", and "123456" are in the top 10 most common passwords), an attacker with access to the database can simply compare the stored hash codes to a dictionary of hash codes computed from common passwords. This won't compromise all user accounts, but almost always it will compromise at least a few accounts.

To prevent dictionary attacks, one can prepend a *salt* value to the password before applying the hash function; if the salt is large enough then the potential for dictionary attacks is eliminated. Of course, there is still the issue of users picking bad passwords, but several techniques can prevent

brute-force attacks, such as imposing a time delay on authentication failure, and enforcing strict policies on password choices such as expiring passwords after a period of time, and disallowing users from using any common password.

For this section, you will implement a secure password storage mechanism using both hashing and salting. Note that such functionality is normally implemented in the application sitting on top of the database, or even in the web client. You, however, will be implementing these operations as stored routines in the database, simply to avoid the complexity of having to interface with the database from another programming language.

All of your work for this problem will be in the `setup-passwords.sql` file, which we've started for you. Additionally, a helper function called **`make_salt`** is provided for you at the top of the file, which is able to generate a random salt of up to 20 characters (Note: Depending on your MySQL version, you may run into an error message about **`NOT DETERMINISTIC`**; it is alright to change this to **`DETERMINISTIC`** if you run into this). You can use it like this:

```
SELECT make_salt(10) AS salt;
```

First, we've provided with you a table table definition **`user_info`** that will store the data for this password mechanism. There are three string values the table stores:

- Usernames will be up to 20 characters long
- Salt values will always be 8 characters
- The hashed value of the password, called **`password_hash`**

We will be using the SHA-2 function to generate the cryptographic hash. (MySQL also has the MD5 and SHA-1 hash functions, but both of these are now considered too weak to be secure.) You can try out the SHA-2 hash function in MySQL like this:

```
SELECT SHA2('letmein', 256);
```

This built-in function can generate hashes that are 228 bits, 256 bits, 384 bits, or 512 bits. We will use 256-bit hashes. Note that the hash is returned as a sequence of hexadecimal characters, so you will need to choose a suitable string length to store the hexadecimal version of the 256-bit hash. As a result, the **`password_hash`** column is a fixed-size column that is exactly the right size, no larger and no smaller.

Alright, with all that said, we'll leave the rest of the work to you! Finish the two procedures in the provided code; do not change the **`-- [Problem 1x]`** labels.

1a) Finish the stored procedure **`sp_add_user(new_username, password)`**. This procedure is very simple:

- Generate a new salt.
- Add a new record to the **`user_info`** table with the username, salt, and salted password.

Make sure that you prepend the salt to the password *before* hashing the string. You can use the MySQL **`CONCAT(s1, s2, ...)`** function for this.

Don't worry about handling the situation where the username is already taken; assume that the application has already verified whether the username is available, though you can improve this procedure to support that for an added challenge.

1b) Finish the function (not a procedure) called **`authenticate(username, password)`**, which returns a **`TINYINT`** value of **`1`** (true) or **`0`** (false) based on whether a valid username and password have been provided. The function should return **`1`** iff:

- The username actually appears in the **`user_info`** table, and
- When the specified password is salted and hashed, the resulting hash matches the hash stored in the database.

If the username is not present, or the hashed password doesn't match the value stored in the database, authentication fails and **`0`** (false) should be returned.

Note that we don't distinguish between an invalid username and an invalid password; again, this is to keep attackers from identifying valid usernames from outside the system.

1c) Of course, you'll need some users for us to test with! Add at least 2 users using your **`sp_add_user`** procedure **at the bottom of this file** so that when we load it, the database will be set up with users that your application will likely be interfacing with. As mentioned above, you may adjust the schema of the **`user_info`** table to support a fourth attribute to represent the user's status, depending on the design of your application.

1d) Create a stored procedure **`sp_change_password(username, new_password)`**. This procedure is virtually identical to the previous procedure, except that an existing user record will be updated, rather than adding a new record. Make sure to generate a new salt value in this function!

Testing Your Code

Once you have completed these operations, make sure you test them to verify that they are working properly (these are also included in the provided **`pw-tester.sql`** file on Canvas). For example:

```
CALL sp_add_user('alex', 'hello');
CALL sp_add_user('bowie', 'goodbye');

SELECT authenticate('chaka', 'hello');     -- Should return 0 (false)
SELECT authenticate('alex', 'goodbye');    -- Should return 0 (false)
SELECT authenticate('alex', 'hello');      -- Should return 1 (true)
SELECT authenticate('alex', 'HELLO');      -- Should return 0 (false)
SELECT authenticate('bowie', 'goodbye');   -- Should return 1 (true)

-- provided tests for optional 1d
CALL sp_change_password('alex', 'greetings');

SELECT authenticate('alex', 'hello');      -- Should return 0 (false)
SELECT authenticate('alex', 'greetings');  -- Should return 1 (true)
SELECT authenticate('bowie', 'greetings'); -- Should return 0 (false)
```

Part B: Final Project Milestone

After you finish Part A, you will start incorporating what you've learned so far towards your Final Project. Based on past term feedback, we are incorporating time for the Final Project earlier, and **you should expect to spend 3-4 hours this week on Part B.**

Requirements for Sunday 11:30PM Milestone:

- If you have not checked in with El about your project feedback, you will need to by EOD Saturday (see Calendly or request group OH)
- Finalize your DDL and submit to CodePost as setup.sql
    - We will be learning more about ER diagrams on Thursday, which we recommend getting started on refining as well (some of you already have a great start!), though not required for this week
- Start your app-client.py/app-admin.py with a menu interface and function stubs for the different tasks of your program; you may alternatively use app.py with a separate get_conn() for admin vs. client permissions, but students in the past have had stronger submissions with a clear distinction between the two programs
    - We have provided an app-template.py on Canvas as an example starter program; most students who break this down into a client program and admin program tend to have an easier time with the overall project flow (you may rename them as you'd like)
    - Use your UI diagrams and feedback from El to guide your program decomposition (you aren't required to implement functions for full credit, though it's recommended you get some start)
    - Use #fp-milestone-discussion on Discord to continue collaborating with peers; engagement on this channel will be considered in the milestone
- Any notes from your proposal meetings, team meetings, peer feedback, etc. in a progress-report.txt or progress-report.pdf (at least 2-3 sentences, but the more the better)
- You can find a preview of the Final Project specification here (subject to minor updates for 24wi summarized at the introduction)