Task 6 : Cryptographic Failures (Supporting Material 1)

The most common way to store a large amount of data in a format easily accessible from many locations is in a database. This is perfect for something like a web application, as many users may interact with the website at any time. Database engines usually follow the Structured Query Language (SQL) syntax.

In a production environment, it is common to see databases set up on dedicated servers running a database service such as MySQL or MariaDB; however, databases can also be stored as files. These are referred to as "flat-file" databases, as they are stored as a single file on the computer. This is much easier than setting up an entire database server and could potentially be seen in smaller web applications. Accessing a database server is outwith the scope of today's task, so let's focus instead on flat-file databases.

As mentioned previously, flat-file databases are stored as a file on the disk of a computer. Usually, this would not be a problem for a web app, but what happens if the database is stored underneath the root directory of the website (i.e. one of the files accessible to the user connecting to the website)? Well, we can download and query it on our own machine, with full access to everything in the database. Sensitive Data Exposure, indeed!

That is a big hint for the challenge, so let's briefly cover some of the syntax we would use to query a flat-file database.

The most common (and simplest) format of a flat-file database is an SQLite database. These can be interacted with in most programming languages and have a dedicated client for querying them on the command line. This client is called `sqlite3` and is installed on many Linux distributions by default.

Let's suppose we have successfully managed to download a database:

Linux

```
user@linux$ ls -l
-rw-r--r-- 1 user user 8192 Feb  2 20:33 example.db
```

```
user@linux$ file example.db
example.db: SQLite 3.x database, last written using SQLite version 3039002, file
counter 1, database pages 2, cookie 0x1, schema 4, UTF-8, version-valid-for 1
```

We can see that there is an SQLite database in the current folder.

To access it, we use sqlite3 <database-name>:

```
user@linux$ sqlite3 example.db
SQLite version 3.39.2 2022-07-21 15:24:47
Enter ".help" for usage hints.
sqlite>
```

From here, we can see the tables in the database by using the `.tables` command:

```
user@linux$ sqlite3 example.db
SQLite version 3.39.2 2022-07-21 15:24:47
Enter ".help" for usage hints.
sqlite> .tables
customers
```

At this point, we can dump all the data from the table, but we won't necessarily know what each column means unless we look at the table information. First, let's use `PRAGMA table_info(customers);` to see the table information. Then we'll use `SELECT * FROM customers;` to dump the information from the table:

```
sqlite> PRAGMA table_info(customers);
0|cudtID|INT|1||1
1|custName|TEXT|1||0
2|creditCard|TEXT|0||0
3|password|TEXT|1||0sqlite> SELECT * FROM customers;
0|Joy Paulson|4916 9012 2231 7905|5f4dcc3b5aa765d61d8327deb882cf99
1|John Walters|4671 5376 3366 8125|fef08f333cc53594c8097eba1f35726a
2|Lena Abdul|4353 4722 6349 6685|b55ab2470f160c331a99b8d8a1946b19
3|Andrew Miller|4059 8824 0198 5596|bc7b657bd56e4386e3397ca86e378f70
4|Keith Wayman|4972 1604 3381 8885|12e7a36c0710571b3d827992f4cfe679
5|Annett Scholz|5400 1617 6508 1166|e2795fc96af3f4d6288906a90a52a47f
```

We can see from the table information that there are four columns: `custID`, `custName`, `creditCard` and `password`. You may notice that this matches up with the results. Take the first row:

0|Joy Paulson|4916 9012 2231 7905|5f4dcc3b5aa765d61d8327deb882cf99

We have the custID (0), the custName (Joy Paulson), the creditCard (4916 9012 2231 7905) and a password hash (5f4dcc3b5aa765d61d8327deb882cf99).

In the next task, we'll look at cracking this hash.

Answer the questions below :

1. Read and understand the supporting material on SQLite Databases.
A. No answer needed