CS-C3130 Information Security (2021)
Exercise 2

SQL Injection and Password Cracking

This exercise covers two topics. First, in parts A and B, your goal is to exploit an SQL query vulnerability in a web application. As the result of this exploit, you should be able to retrieve the password hashes stored in the SQL database on the server. Then, in part C, you will crack some of the passwords from the hashes.

The attack target is the Potplants web application, which is used for storing and sharing information about plants. Launch the application from the exercise launcher and log in with the username **student** and the password provided in the exercise launcher. The exercise launcher is at https://infosec1.vikaa.fi/.

Part A. Simple SQL injection (10 p)

The Potplants application has a plant search feature. When the user enters a search string, the browser sends it to the server, which queries the database for the requested plant. This is how the SQL query is composed in the `node.js` code on the server:

```
var sql_query = 'SELECT id, name, color, planttype, potsize, shared FROM
    plants WHERE user_id=' + user_id + ' AND name LIKE "%' + plant + '%";';
```

The variable `user_id` contains the ID of your user account, and `plant` is the search string that you entered. String concatenation for composing SQL statements is bound to lead to vulnerabilities. Any responsible programmer like you would, of course, use *prepared statements* instead.

Your first task is to retrieve `plants` data that belongs to other users. To achieve this, you need to be clever about the search string. The exploit string in this case can be very short. (Fun challenge: what is the shortest possible exploit string?)

Note that, if the SQL query syntax is wrong, an exception is raised in the server. This happens, for example, when you try to access a non-existent table or column name, when there is a type error in evaluating the query, or when the quotation marks are not closed or the semicolon is missing from the end of the query. Since SQL error messages often include information that is helpful to hackers, the Potplants application does not display the error messages to normal users. In some cases that were not anticipated by the programmer, it does not even handle the exceptions properly and no response may be returned. On the other hand, when the query is successful, some (or zero) rows of data are returned and shown to the user. The first step in SQL injection attacks is to try to inject any SQL code that runs without errors. After that, you can gradually edit the query to return some interesting data. *This is one of the biggest challenges in hacking: the hacker is forced to develop and debug the exploit code blindly, with only vague hints about whether each attempts is successful or not, and why it is not working.*

To submit your solution to this part, copy one plant name from the *other* users of the Potplants site to the exercise launcher.

Part B. SQL injection to steal password hashes (10 p)

Your goal in this part of the exercise is to execute an SQL injection attack to retrieve password data, i.e., usernames, salt values and password hashes, from the Potplants web site.

You can exploit the same vulnerability as above, but the task is slightly more complicated because the password data is stored in a different table and because you need to first discover the names of the relevant table and columns. You can find ideas from the lecture slides and SQL injection tutorials online.

Hints: The method for discovering table and column names varies greatly between SQL servers. The exercise uses `sqlite3` where the query `SELECT * FROM sqlite_master;` would return the relevant information (see the sqlite3 documentation). Try to include something similar in the injected SQL to retrieve the table and column names. When you know the table and column names, repeat the process to retrieve the password data.

When you have retrieved the other users' password data, submit your solution by copying one user's data to the exercise launcher.

Part C (bonus problem). Password cracking (10 p)

Once you have retrieved the usernames, salt values and password hashes, you naturally want to crack the actual passwords. Luckily for you, some of the passwords are relatively weak, such as words found in a dictionary. However, most all are not quite so easy to guess.

From open-source code or by reverse engineering, you have discovered that the password hash for each user is computed from the concatenation of three strings: the constant string "potPlantSalt", the password, and the salt value. The SHA-256 algorithm was used for hashing the passwords. (An iterated hash such as PBKDF2 or Argon2 should be used to make the password cracking harder.) The SHA-256 output has been converted into hexadecimal format and truncated to 32 characters before storing in the database as a string.

```
truncate(hexstring(SHA256("potPlantSalt" + password + salt)))
```

You should try to crack 1-10 passwords (one point for each password). You may write a script or use an existing password-cracking program of your choice to perform the attack. Note that while you always can crack some passwords, it is impossible to crack them all.

In case you decide to write the cracking program by yourself, the Python library `hashlib` or the C library function `mhash(3)` may be useful. Use `student`'s password as the test case for debugging your code. You can find word lists under `/usr/share/dict` in a typical Linux distribution and also on `brute.aalto.fi`. In addition to trying just the dictionary words, you should think about possible mutations and combinations that people might apply to make their word-based passwords harder to guess.

The best place to do the password cracking is on your own computer as it requires more informed guesswork than brute computing power. Aalto has two servers which are intended for light computation tasks: `brute.aalto.fi` and `force.aalto.fi`. You can `ssh` to these servers and run your program, but please use only one CPU core at a time and limit the runtime to one hour before modifying your code and trying something else.

Supplemental reading

- OWASP: SQL Injection
- OWASP SQL Injection Prevention Cheat Sheet