Graham Dungan
January 28, 2025
UIN: 332001764

CSCE 413: Software Security
PoC 4

Development of Vulnerable Application (30 Points)

For this assignment, I have created a simple Flask banking app that allows the user to check their account information by entering their account ID. The entry field for the account ID is where SQL injections can be performed. To run both vulnerable and secure versions of the web application, please follow these steps for Linux;

1. Enter the `PoC4` folder.
   `cd PoC4`
2. Create a Python virtual environment.
   `python3 -m venv .venv`
3. Activate the virtual environment.
   `source ./.venv/bin/activate`
4. Install all requirements.
   `pip install -r ./requirements.txt`
5. Run either the vulnerable or secure versions of the program.
   `python3 ./vulnerableWebapp.py`
   `python3 ./secureWebapp.py`

The application is hosted on the route `localhost:5000/login` and provides a simple login form. Once the user has entered a valid `user_id`, they will be shown their ID, name, password, and balance. The database consists of a single `users` table that contains all of this user information;

USERS	
PK	user_id INT
	name VARCHAR(255)
	password VARCHAR(255)
	balance DECIMAL(10,2)

While somewhat unrealistic, this is a demonstration of how manipulating SQL queries can result in sensitive information being leaked. What follows is a brief explanation of the application and why it is vulnerable.

SQL Injection Vulnerability

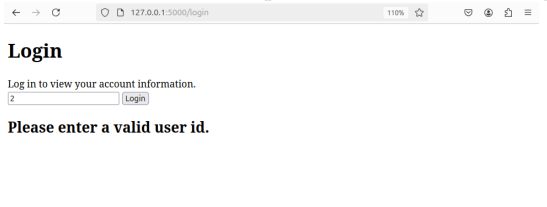
The portion of the code that leaves the web application vulnerable to SQL injection attacks can be found on lines 79-82 of `vulnerableWebapp.py`,

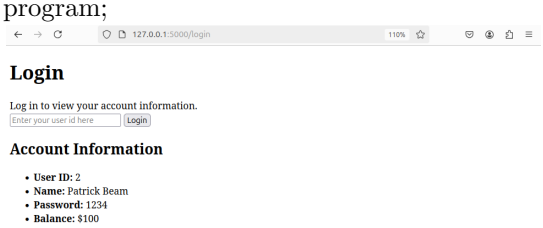
```
1 query = f"SELECT␣*␣FROM␣users␣WHERE␣user_id␣=␣{user_id}"
2 cursor.execute(query)
```

This method of executing queries is vulnerable to SQL injections because it does not parse the `user_id` user input before joining it with the query f-string. Since `user_id` is a string, any other information within it could be interpreted as another part of the query. For example, since this f-string is essentially concatenating any string provided as `user_id`, if `user_id="2 OR 1=1;"`, the query string would become `query="SELECT * FROM users WHERE user_id = 2 OR 1=1;"` resulting in a SQL injection attack.

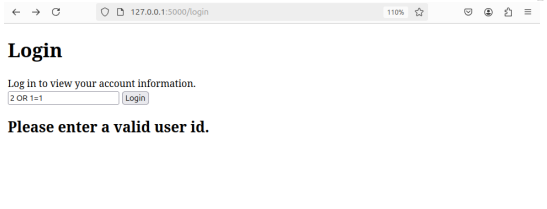
Exploitation of SQL Injection (20 Points)

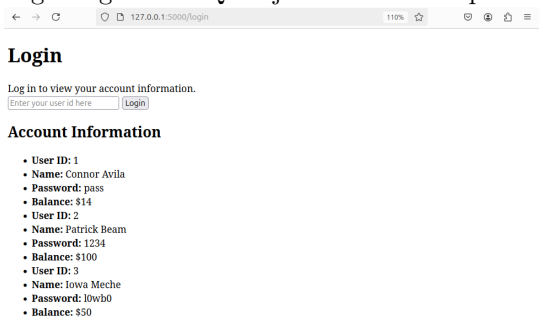
An SQL injection can be performed in `vulnerableWebapp.py` if the user provides any string that extends the query parameters. A simple example of this attack is providing the string `"2 OR 1=1;"`. As discussed earlier, this will result in a query statement that selects all users since the `WHERE` portion of the query is always true, given `1=1`. What follows is the expected execution of the program;



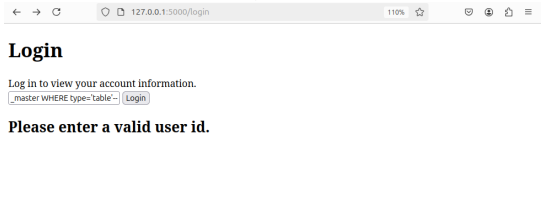


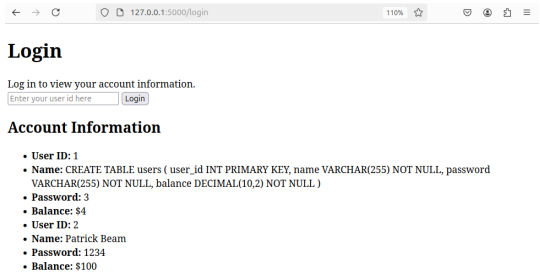
Now we will demonstrate the behavior of the program given a SQL injection that dumps all user information;





Following, `UNION` statements can be used to learn more information about the structure of the database, as referenced here. An example of an injection string would be;
`2 UNION SELECT 1, sql, 3, 4 FROM sqlite_master WHERE type='table'--`
as demonstrated here,





Implications

Given that any string can be (effectively) concatenated to the query string, any statement modifying the truth of the `WHERE` selection or including `UNION` could be successful. It should be noted, though, that SQLite `execute()` statements cannot execute multiple statements, as seen in the SQLite3 documentation. Given this, it would not be possible to execute table manipulating strings in this query unless the cursor was able to execute multiple statements (possibly with a function like the `executemany()`, as mentioned here).

Mitigation of SQL Injection (20 Points)

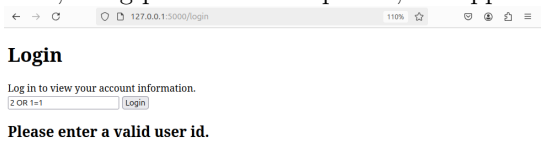
SQLite offers the ability to parameterize queries such that user-provided information is treated only as data and is not recognized as a statement. What follows is a code snippet of the now safer SQL query execution, preventing a SQL injection;

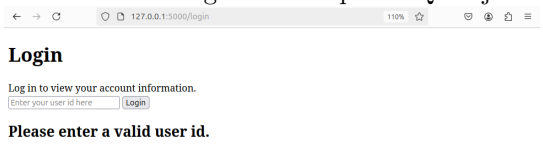
```
query = f"SELECT␣*␣FROM␣users␣WHERE␣user_id␣=␣?;"
cursor.execute(query, (user_id,))
```

It should be noted that SQLite does not offer stored procedures or prepared statements. It would be possible to create something akin to a stored procedure by moving the database connection and query execution commands to a separate function. SQLite's parameterized queries act effectively as prepared statements, as seen by the discussion found here.

Demonstration of Fix Effectiveness (20 Points)

Now, using parameterized queries, the application does not recognize attempted SQL injections.





As stated earlier, this attack is no longer effective because all user input is used in a parameterized query, meaning no user-supplied data can be run as code. See the forum post referenced in the section **Mitigation of SQL Injection** as to why this is equivalent in security to a prepared statement.

Documentation (10 Points)

See above.