Blind SQL Injection

Overview:

Blind SQL injection is the type of attack wherein, information is extracted from the database through a series of true/false questions. The critical aspect of any Blind SQL injection attack is examining how the "to be attacked website" behaves on different inputs. In simple terms, instead of asking questions like, what's the password stored in the database we ask questions like, "is the first character of the password 'a'? Based on the application's response we either change the question or move to the next character.

Goal:

In any web application, most of the User data is stored in databases for authentication and authorization. Majority of the authentication based applications rely on password associated with the User stored in the database. The tool attempts to extract critical information from the database that the vulnerable application uses to store data. Since database error messages are suppressed by the application we rely on success and failure messages thrown by the application to decide on the correctness of query execution.

Building Blocks:

- **Information for the tool:**
  - The tool takes the following input:
    - Vulnerable application URL (http://<domain>:<port>).
    - Post/GET parameters in key, value pair format.
    - Message application throws on successful execution.
    - Message application throws on failed execution.
    - Type of request the form makes (GET, POST).
- **Attack Query:**
  - The attack query is the most critical part of performing a blind SQL injection attack. The most basic form of this query will be,
    1=if (query for information) <condition> <value>, 1, 2.
    The above frame returns true when the condition evaluates to true and returns 1.
- **Injecting Attack Query:**
  - Most web applications take form input from the client which is then processed at the back end. Understanding those request parameters and their structure will reveal the point of the attack. It is usually the parameter that is used to make database queries.
  - In our tool we make the assumption that the parameter used to make back end query will either have the "name" or "user" substring.
  - The desired structure of the final query "select <field> from <table> where <field_value>=<parameter_value>" should look like
    "select<field> from <table> where<field_value>=<parameter_value> and 1=if (information extraction query) <condition> <value>, 1,2)
  - Now, if the expression in the condition is turns out to be true, the query will return a result and we can verify that through the existence of the success message.

- **Binary search:**
  - While asking binary questions through the Attack Query as explained earlier we basically perform binary search for the actual value.
  - For e.g. if the tool wanted the length of the output of certain query, it would embed the "select length ((query))" in the expression of the "if" construct. Then we will search of the value using binary search using "=, >, <" in the <condition> placeholder.
- **Content Extraction:**
  - Content extraction query is a two step process
    1. Get the length of the query output using "select length ((query))".
    2. Get the actual output of the query using "select ord ((select substring ((query), i, 1)))" where i goes from 1 to length (result of query).

    The ord () function returns the ASCII value which is again searched using binary search.

Approach:

- **Extract Schema Name:**
  - The schema name is used to get data such as the column names from the information_schema table that has the Metadata for every schema.
  - Every attack follows the same sequence of finding the length and then extracting the content character by character.
  - The query used to extract schema name is "select schema ()".
- **Extract Table Names in Schema:**
  - Table names are required to construct queries for data extraction and also get column names.
  - First step is get the number of tables using the query "select count(table_name) from information_schema.tables where table_schema=<schema_name>".
  - The query used to get the table names is "select table_name from information_schema.tables where table schema=<schema_name> limit i,1" where i goes from 0 to number_of_tables-1.
- **Extract Column Names in Tables:**
  - To get contents from tables, it is essential to get column names since the query being executed inside the if constrict of the attack query doesn't execute properly if the result is not single valued.
  - The information extraction for this part begin with first getting the number of columns using the query "select count(column_name) from information_schema.columns where table_name=<table_name> and table_schema=<schema_name>".
  - The query to get the column names is "select column_name from information_schema.columns where table_name=<tableName> and table_schema=<schema_name> limit i,1" where i goes from 0 to number_of_columns – 1
- **Extract Data from Tables:**
  - The tool first gets the number of rows using the query "select count(*) from <table_name>".
  - Once column names and tables names are extracted, content extraction is done through the query "select <column_name> from <table_name> limit i,1" where i goes from 0 to number_of_rows-1.

Limitations:

- The tool will work only if the input parameters are not sanitized. Any form of modification made to the parameter values to suit the requirements of the application will fail to construct the correct SQL attack queries.
- Non ASCII characters will not be printed since the binary search range is from 0 to 127.
- The point of injection is determined by analyzing the parameter names. If the names are obscured, then discovering the attack point might need enhanced analytical behavior from the tool.
- If the password is stored in hashed form, other forms of attacks like dictionary or rainbow table attacks might become necessary to get passwords.
- Sometimes the order of VARCHAR type columns might not be the way it is stored in the database.

Test Case Execution and output:

- Analyzing the vulnerable application form, the parameter input to the application could be formed.
- Inspecting the application behavior it was clear that it displays the file content, so we can get the success message by uploading a file with the message we want.
- Once we have these things in place, run the tool with the following input "http://localhost:8080/test1.php" "name,Tarun Shimoga;password,abcd;submitaccess,1" "Warning" "Error" "post".
  The output looks like:

```
**************************
Schema Name: test
Schema has the following tables:
files user
Content from table: files
Table has  3 rows
Tarun Shimoga abcd
Manit Singh manit
Deepesh Mittal deepesh
Content from table: user
Table has  1 rows
tshimoga tarun shimoga tarun tshimoga@asu.edu
**************************
```

Future Enhancements:

- The tool can be made to detect injection points by analyzing the application behavior. The most basic approach would be to brute forcibly treat every parameter as injection argument and run the tool few times to check for extracted data.
- Robustness can be added by exploring and understanding sanitizing approaches and finding loop holes.
- The tool can also be made to handle GET or POST requests without the need for giving them as arguments.