67607 – Exercise 4 – Blind SQL Injection

Instructed by Amit Klein, written by Agam Ebel

2025

Contents

1 Blind SQL injection

1.1 Introduction

SQL injection (which from now on we will refer to as SQLi) is a code injection technique where one can place malicious code in SQL statements via web page input.
There are several types of SQLi. This exercise will demonstrate the "boolean-based **blind SQLi**" variant.

Your goal is to extract a password from the "users" table in the database and print it to a TXT file (the text file should be named with the ID of the one who submits the files). Notice that the password should correspond to the ID of the student who actually submits the files on Moodle.

1.2 Our Environment

Our web application is vulnerable to blind SQL injection and deployed within a docker container. The container is built by installing the necessary packages and configuring the web application and the web server.

We also have a MariaDB SQL server (an offshoot of the venerable MySQL server) deployed within another docker container. It will be used as our database.

The attacker is deployed within its own docker container, its goal is to extract a password (which corresponds to a specific ID) from the "users" table from the database.

All the containers are connected to a virtual LAN named `bsqli-net`, this is an isolated network where we operate.

1.3 General Information

- To log in to the environment, use username "bsqlEnv" and password 5260.
- All the containers are connected to a virtual LAN named `bsqli-net`
- The database is a MariaDB database (similar to MySQL database) and deployed by a docker container. It is connected to a virtual LAN and its IP address is `192.168.1.201`.
- The web application is connected to the database and deployed by a docker container. It is connected to a virtual LAN and its IP address is `192.168.1.202`.
- The attacker is deployed by a docker container. It is connected to a virtual LAN and its IP address is `192.168.1.203`.
- We can access the web application (from the containers) by creating a request to `http://`*`web-application-IP-address`*`/`

(Notice that for accessing the web application from the VM we need to use "localhost" instead of the proxy server IP and port 8080, i.e. creating a request to `http://localhost:8080/`)

- There is a table named "users" on the database which contains 2 columns named "id" and "password". This is the table the attacker wants to extract the information from. Note that this is **not** the table queried by the web application to get its "production" data.

- You can connect to the database and learn the table schemes in the following way:

  1. connect to the container using `docker exec -it mariadb-server /bin/bash`
  2. connect to the database using `mariadb -u u67607 -p 67607db` and insert the following password `courseUser`.
  3. Now you can send `SELECT` queries to see the data saved in the tables (and also insert data to check your attack code, using `INSERT INTO...` statement).
     Note, to see the values in the "password" column, use the following query (we encompass the password string between "<" and ">" symbols to clearly delineate where the password string begins and ends):

     ```
     SELECT id, CONCAT("<", password, ">") AS pwd
     FROM users;
     ```

  Note that the password may also contain those delimiters (<,>).

  Please note that in the grading environment, the ability to modify or add records to the database is disabled. This functionality is enabled here solely for the purpose of testing your code.

- The "users" table you get in the exercise will not have the same values in the password column as the one in our testing environment.

- The "id" is 9 digits (i.e. 123456789) including leading zeros, if they exists in the ID.

- The password consists of printable ASCII characters (`0x20` - `0x7F`) and its length is 1-10 characters.

- You can find the source code of the application in the following path: `/home/bsqlenv/Desktop/index.php`.

- There are 2 valid values in the "orders" table, one is with `order_id=1` and the other with `order_id=2`, run a query and ask for them, you can see that they return "The order has not been sent yet" and "The order has been sent".

- In the "orders" table, "0" is not a valid value for `order_id`.

1.3.1 Notes About Our Web Application

We have a PHP web page that will use us as the vulnerable web application. This is a simple dynamic page that will take information from the database, and according to the data provided shows a message to the screen. We can access the web page from the attacker container or via the VM.

The page is called `index.php` looks like this:

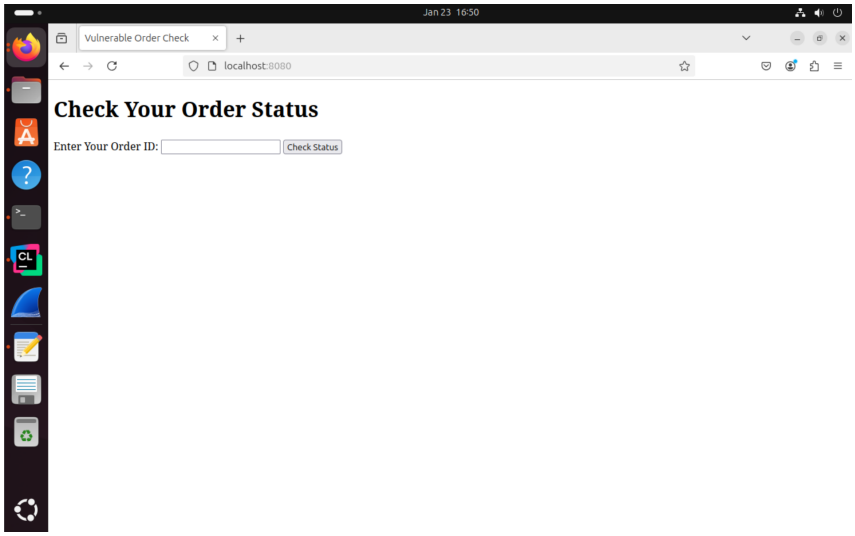

Figure 1: Our web application

When the text message is "Your order has been sent!" if the answer to the query is not null or 0. Otherwise, the text message is "Your order has not been sent yet."

You can enter values to the `order_id` and see what message you get and see in the URL in which parameter you inserted the data (as the application uses the `GET` method for sending the page to the web server).

1.3.2 Notes About SQL Queries

SQL (Structured Query Language) is used to store, manipulate, and retrieve data in a structured format from a relational database.
In this section, we will provide a brief introduction to some important SQL query syntax:
`SELECT` – This is the basic command we need in SQL queries to retrieve data. It tells the database which columns we want to get the data from.
`FROM` – This is a basic clause we need in SQL queries to retrieve data. It tells the database which table to retrieve the data from.
`WHERE` – We can use this clause to filter which rows will return based on specific conditions.

For example, assume that we have a table named "Students" which has 4 columns: first_name, last_name, course, and degree_type (graduate/undergraduate student).
We want to get all students who learn the course 67607. We can do that by sending the following query:

```
SELECT first_name, last_name
FROM students
WHERE course = 67607;
```

Note that a complete SELECT statement must have the following structure: SELECT *(columns, expressions)* FROM *tables* WHERE *condition*;
When the WHERE part can be optional.

Next, we will introduce some important operators in SQL. We can divide them into different kinds, for example, logical operators are used in the "condition" part, and logical expressions and operators which operate on complete query results.

Some examples of a logical operation are AND, LIMIT and LIKE:

- AND – This operator is used to combine multiple conditions inside a WHERE clause.
  Recall the table from the previous example. We want to find all students who are undergraduate students and learning the course 67607, we can do it by the following query:

  ```
  SELECT first_name, last_name
  FROM students
  WHERE degree_type='undergraduate' AND course = 67607;
  ```

- LIMIT n,k – This operator (more accurately: clause) limits the number of rows returned in the result set – only k rows will be returned, starting at row n.
  Recall the previous example. We want to limit our search to 1 student at a time who is an undergraduate student and learning the course 67607, we can do it by the following query:

  ```
  SELECT first_name, last_name
  FROM students
  WHERE degree_type='undergraduate' AND course = 67607
  LIMIT k,1;
  ```

  where k tells us the offset to take the result from (i.e. to take the value in the k-th row).

- `LIKE` – This operator lets us match a pattern in a column. The pattern is similar to a regular expression.
  Recall the "students" table, we can find all students whose names start with "A" by using the following query:

```
SELECT first_name, last_name
FROM students
WHERE first_name LIKE 'A%';
```

  Here, the `%` in the expression means match anything (0 or more characters, like * in regular expressions).

Lastly, we have `UNION` which operates on complete query results:
`UNION` – This operator is used to combine the results of 2 or more `SELECT` statements. Notice that we do not have to select the data from the same tables, but the columns returned must have the same data type, size, and the same order.
For example, assume that we have a table named "CS-students" and a table named "EE-Students" (as a shorthand for electrical engineering students). Both of them have 4 columns: first_name, last_name, course, and degree_type (graduate/undergraduate student). We want to get all students who learn the course 67607. We can do that by sending the following query:

```
SELECT first_name, last_name
FROM cs-students
WHERE course = 67607
UNION
SELECT first_name, last_name
FROM ee-students
WHERE course = 67607;
```

More information about SQL queries and operators can be found here `https://www.geeksforgeeks.org/sql-cheat-sheet/`
In case you want to learn more about SQL or not familiar with SQL, we recommend you to read this tutorial.

1.3.3 Boolean Based Blind SQL Injection Reminders

To recall, in contrast to other types of SQL injection, in blind SQL injection, we don't see a direct output of an SQL table.

There are 2 types of blind SQL injection: "boolean-based" and "time-based". This section will briefly introduce the "boolean-based blind SQL injection" concept.

In this attack type, some information about the SQL query output can be inferred by the data displayed after the query has been made (i.e. we see if an item is "in stock" or "out of stock" instead of 0 or null for items out of stock). This represents a single bit of information about the query output.

1.4 Starting the environment

The environment for the attacks consists of a VM and 3 containers. First, download the environment at the following link.
Note – the environment disk requires at least 30GB free on your computer, make sure to have enough free space to upload the disk.
Start the virtual machine using the instructions for uploading a virtual disk to a VM which we provided in the "Environment Setup" file.

Next, start the containers:

1. Start the MariaDB SQL server using `docker start mariadb-server`.
2. Start the web application using `docker start web-app`.
3. Start the attacker's container using `docker start attacker`, next run `docker exec -it attacker /bin/bash`.

Note that you can work on your source code file from within the VM (you should place your files under the directory `/home/bsqlenv/Desktop/attacker` on the VM), and you'll be able to access it in the attacker's container under `/tmp/attacker` (which is mounted to the VM's `/home/bsqlenv/Desktop/attacker`).

Therefore, it is enough to change the source code itself inside the "attacker" directory on the VM and the changes will also applied in the container. You must compile the source code from inside the container (with the following command: `gcc -Wall -Wextra -Werror -Wconversion` *`attack-file.c -o attack-name`*) and ensure that there are no compilation warnings, and run the executable from the container. This is how we will compile and run your code ourselves (in our container environment which is similar to yours), so this guarantees you're working with the same compilation and runtime environment we use. We will take off points for compilation warnings (and lots of points if it fails to compile or do the job).
Note that we added the flag `-Wconversion` to the compilation flags.

1.5 Tips and Tricks

- If we try to exploit the vulnerability using a payload that contains `UNION SELECT` in the injected query, we need first to "get rid" of the result of the original query.

- We recommend to first find which query the app uses to interact with the database, insert data and see how it behaves. Next, try to understand how inject into it, check that the injection works and finally, write the attack code.
- Check your code! Run your attack attempt several times to ensure everything works as expected. We recommend running your code numerous times, ideally dozens of times (as this reflects how we will evaluate it).
- Do not forget to encode spaces as `%20` (and other characters i.e. "=" and "?") when you send your attack payload as a part of a request for the page URL.
- 2 useful ways to add your payload to the original query are using sub-queries and using the `UNION` operator.
- Make sure your **implementation** is as simple as possible, i.e. does things efficiently and has no unnecessary code/logic. An unnecessary complex, twisted or superfluous logic takes us longer to grade and we will take off some points for it.
- It is recommended to look at the traffic in order to understand if the queries aren't malformed, as you may not get a detailed error message from the web application.
- You can find the arithmetic operators that MariaDB supports in the following link: `https://mariadb.com/kb/en/operators/`
- You can find the string functions that MariaDB supports in the following link: `https://mariadb.com/kb/en/string-functions/`
- It is recommended to restart the web application container between one experiment and another. We can do it by using `docker restart web-app`.
- Pay attention to the number of queries you send, and do not send unnecessary queries (i.e. duplicate queries, as it might exceed the 100 queries limit).

1.6 What to Submit

1.6.1 Important Remarks

Please follow these instructions carefully to ensure full credit for your submission. Failure to meet any of these requirements will result in a deduction of points.

- Note that your code should interact exclusively with the web application and must not directly access the database.
- Note that the password may contain spaces at the start/end. We will check for exact matches of the password when we evaluate your code.

- **Do not send more than 100 queries**. Yes, this is a strict and very limiting restriction. You need to ensure that your queries are efficient (query-count-wise).
- You are required to write the password you get from the database to a `TXT` file.
  The text file should be named with the ID of the one who submits the files in Moodle (i.e. 12345689.txt) and should contain the password from the database for the corresponding ID in the following format: `*password*` (note the asterisks).
  Note, do not add the <,> delimiters when you print the password into the file.
- When compiling your attack files, use `gcc -Wall -Wextra -Werror -Wconversion` *`program-name.c -o desired-compiled-file-name`*
  Compile your source code from within the attacker's container and make sure there are no compilation warnings! We will take points off for any compilation warnings.
  Note that we added the flag `-Wconversion` to the compilation.
- When running the compiled code, run it without any command line arguments.
- Make sure your code does not print anything to the screen before submitting your attack attempt. Except in cases of failures (i.e. failure in an opening socket) or unexpected scenarios, then you can print to `STDERR` (i.e. using `perror` or `fprintf(stderr,...)` and terminate the program with `exit(1)`.
- The exit code from the attack code should be 0 (not including failure cases, in this case, use `exit(1)` for terminating the program).
- Please clean up your code before submission – remove redundant code, add useful comments, use reasonable names for variables and functions, etc. – but please double-check that everything still works afterwards!

1.6.2 Submission

The submission is done via Moodle in pairs. Please submit a **single zip file named "ex4.zip"** which contains the following files in a "flat" format (i.e. not under a folder/directory):

1. Your attack attempt code (in `C`). The file should be named "ex4.c".
   When compiling the file, use `gcc -Wall -Wextra -Werror -Wconversion` *`program-name.c -o desired-compiled-file-name`*.
   Compile your source code from within the attacker's container, and make sure there are no compilation warnings! We will take points off for any compilation warnings.
   Note that we added the flag `-Wconversion` to the compilation.
2. A file containing a **short** explanation of your attack attempt. Write 1-2 paragraphs maximum.
   The file should be named "explanation.txt".

3. A TXT file contains a single line with the submitters' IDs (separated by a comma. i.e. 123456789,987654321). If you submit the exercise alone (after getting approval from the course staff), enter your ID only (without a comma, i.e. 123456789).
   The file should be named "readme.txt".

4. Please pay attention to file names (typos...) and do not submit any folders or extra files – just a "flat" zip file with those three files only.

5. **Do not submit** the file you write the password you get from the database.