SQL Injection Attacks Detection & Prevention Techniques

Gülsüm Yiğit and Merve Arnavutoğlu

***Abstract*—SQL Injection Attack (SQLIA) is a type of code injection technique that threatens confidentiality, integrity, and availability of web databases. The attacker mostly exploits incorrectly filtered user inputs such as text fields in web applications and tries to insert malicious SQL statements into a legitimate query via the vulnerable user input. By doing so, the attacker can access, insert, modify, or delete critical information in a database without proper authorization. In this survey, we describe and categorize types of SQLIA, and analyze existing detection and prevention techniques against such attacks.**

***Index Terms*—SQL injection, attacks, cyber security.**

I. Introduction

The rapid growth of internet made web applications one of the most popular communication channels. SQL injection is among the oldest such attacks, but even today stands as a serious threat to confidentiality, integrity, and availability of web databases. Last five OWASP (The Open Web Application Security Project) reports for the top 10 web application vulnerabilities showed that SQL injection is ranked first among other vulnerabilities [1].

In a successful SQL injection attack (SQLIA), the attacker tricks a web application in executing a malicious SQL statement. The malicious statement is, for example, sent via user input fields in web forms. If the application fails to validate the input properly, the malicious statements are injected into legitimate queries and forwarded to the DBMS [2]-[5].

In Section II, we list threats posed by SQLIA in detail. In Section III, we describe how a simple attack is performed in detail. In Section IV, we categorize different types of attacks on relational databases and describe what threat category the attack falls into. In Section V, we survey some defense mechanisms against injection attacks.

II. Threads Posed by SQL Injection Attacks

The harm done by SQLIA can be disastrous because a successful SQL injection can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administrative operations on the database such as shutdown the DBMS, recover the content on the DBMS file system and execute commands (xp cmdshell) to the operating system [4]. We can categorize the threats posed by SQLIA as follows:

Bypassing Authentication: If there is no proper validation on username and password, an attacker can login to the system without knowing the correct username or/and password.

Destroying Integrity: Via injected SQL statements, an attacker may modify/delete data items in the database systems.

Breaching Confidentially: SQL injection may enable an attacker to learn about content that is otherwise not accessible. Depending on the type of the vulnerability, the information leaked can go from a simple boolean result to disclosure of the whole database.

Bypassing Authorization: With a successful injection, an attacker can query any parts of the database bypassing of any access control mechanism [4].

Loss of Availability: For systems that allow function calls to DBMS, an attacker can shut down or crash the database service thus rendering it unavailable to other legitimate users.

III. SQL Injection Attack

We now demonstrate a typical SQLIA on the login page of a vulnerable web application. Suppose that, on the login page, ‘login.php’, we have a submit button and two text boxes; one for the username and the other for the password.

```
<input type="text" id="username"
name="username" value="">
<input type="password" id="password" name="password"
value="">
```

When the user enters the username and password, it will be posted to ‘login.php’ via HTTP_POST method. Application will check if any record exists in, say, the USERS table. In the ‘login.php’, a query will be executed through the user inputs.

```
$username = $_POST['username'];
$password = $_POST['password'];
$query="Select * from USERS
where username='$username' and
password='$password'";
```

Assume that a valid username and password pair is ‘Admin’ and ‘12345’. Once the user enters the following inputs to the two textbox of username and password, the following query will dynamically be constructed.

```
$username = "Admin";
$password = "12345";
$query = "Select* from USERS
where username = 'Admin' AND
password = '12345'";
```

Manuscript received May 25, 2017; revised July 4, 2017.

Gülsüm Yiğit and Merve Arnavutoglu are with the Electronics and Computer Engineering Department, University of Gaziantep, Turkey (e-mail: gulsmyigit@gmail.com, merve.arnavutoglu@gmail.com).

The above query has no problems, it returns all records in the USERS table where username is Admin and password is 12345. If any matching entry is found, the user is authenticated. Now suppose that the attacker injects the following code in the username input field.

```
$username = "'; DELETE *
from USERS; --";
$password = "";
```

Then, we have the following query in the runtime:

```
$query = "Select* from USERS
where username = ''; DELETE *
from USERS; --' AND password = "";
```

Once the above query is executed, DELETE statement will be forwarded to DBMS which will, in return, completely delete 'USERS' table. Note that, in SQL, double dashes (e.g., - -) are used to add comments in an SQL query. Attacker comments out the last appended string by the system. Therefore, the password value is irrelevant and may be set to any string.

IV. SQL Injection Techniques

Depending on the application and the properties of the underlying DBMS, some injections are more effective than others. For example, the injection given in Section III, only works for those DBMSs that support multi statement queries. (Note that the query sent in the previous example has two SQL statements each ending with semicolons.) If we do not have such a DBMS, other forms of injections may still be possible.

In this section, we categorize and present different methods of SQL injections that appears in the literature. We also point out what kind of threat they pose on the database.

Tautologies [3]: As part of this technique, the attacker modifies SQL statements by changing the WHERE clause of the query and using tautological terms with OR operator to get distinct results. The OR operator and the terms are appended to the query in such a way that the query always evaluates to TRUE. These attacks are generally used to bypass authentication control mechanisms or access data which is otherwise not readable.

Let's consider the login example in Section III. If the attacker injects the following text, the resulting query looks like the following in the runtime:

```
$username="' OR 1=1 --";
$query = "SELECT *FROM USERS
WHERE username = '' OR 1=1 --' AND password="";
```

Even if no username and password pair is sent to the DBMS, since 1=1 always evaluates to true, the result set will contain all records in 'USERS' table. If the application uses the size of the result set to authenticate the user, the attacker will pass the authentication without supplying a valid username and password pair.

The same attack can be used to bypass authorization mechanisms as well. Suppose that the app lists the credit cards of an authenticated user with a specific type ('DEBIT' or 'CREDIT') which is inputted via the string variable $type. If the following query is used, the app will list all credit cards belonging to all users:

```
$type = "' OR 1=1 --"
$query = "SELECT creditNo FROM USERS
WHERE type = '$type' AND
userId = $userID"
```

Logically incorrect queries [6]: In this approach, the attacker injects illegal or incomplete SQL statement in such a way that the error message leaks the schema of the underlying database. (Schema of the database is needed to carry out other future attacks.) This attack is possible if the DBMS error messages are allowed to be displayed in the client side.

Suppose the following page concatenates the string value of $t with a query without proper validation [6]:

www.examplewebsite.com/index.php?t=98

And then tries to make a logically incorrect query attack by changing the URL as the follows:

www.examplewebsite.com/index.php?t=98'

When the query is rejected, the default error messages containing debugging information like column names, table names will be returned from the database. In this case, an error message contains the following query in the browser:

```
SELECT* FROM USER WHERE id=98';
```

From the error message, the attacker learns that the name of one of the columns is 'id', and the name of the associated table is 'USER'. Attacker can now use this knowledge to conduct more strict attacks to the database.

PiggyBacked Queries [7]: The attacker poses a threat to the integrity of the data by appending a malicious query to the original query string. The SQL injection example provided in Section III can be categorized as PiggyBacked query where the intended query is extended by a 'DELETE' statement. The goal of the PiggyBacked queries is to extract, add or modify data. Here is a common form of this approach:

```
original SQL Statement + ";" + INSERT
(UPDATE, DELETE, DROP) + <rest of the
injected query>
```

Note that the additional queries are separated by semicolon and this attack vector would be possible only if the database supports execution of multiple queries in one statement.

Let's rephrase the previous example by dropping the table USERS instead of deleting all the data in the USERS table. The attacker injects the following code in the username input field

```
$username = "'; DROP TABLE USERS--";
```

which results in the following SQL statement in the runtime that drops 'USERS' table from the database after execution of this statement.

```
$query = "SELECT * FROM USERS
WHERE username = ''; DROP TABLE USERS--'AND
password = "";
```

In general, by injecting Piggybacked queries, the attacker sends a malicious SQL statement that can execute a 'DROP', 'DELETE', 'UPDATE' or 'INSERT' query to destroy the integrity of the data.

Union Queries [8]: This type of attacks is in SQL manipulation category since it is doing operations on 'UNION SELECT' which injects malicious SQL query patterns to the original safe query to get data related to other tables from database. The purpose of this query is to extract data and to bypass authentication [9]. Here is common form of this attack:

```
original SQL Statement + ";" + UNION SELECT + <rest of
the injected query>
```

The rules for combining two or more statement using union query are as follows [9]:

- Column name and order of columns of queries should be same.
- The data types of the columns on tables in the query should be compatible.

Assume the following query is executed from the database:

```
$id = 1234;
$query = SELECT id,username,phone
FROM EMPLOYEE WHERE id ='$id';
```

This query is going to select all information in EMPLOYEE table where id=1234. But what if the attacker inserts a UNION SELECT statement to id value:

```
$id = "1 UNION ALL SELECT creditCardId
from CREDITCARD_TABLE";
$query = "SELECT id,username,phone FROM EMPLOYEE
WHERE id = 1 UNION ALL SELECT creditCardId from
CREDITCARD_TABLE";
```

The result of the first query will be joined to the result of second query which returns all the credit card user. This approach threats confidentiality of data like personal information,financial information etc..

Stored Procedure [10]: This approach is in function call injection category which is a technique of inserting different database function calls like operating system call. This type of attacks deals with stored built-in functions using SQL injection. Stored procedures run directly on database engine [4].

Once an attacker determines which backend database is in use, stored procedures provided by that specific database can be executed, including procedures that interact with the operating system [11]. In addition, since stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows[11].

Let's consider the same example 'login' system. If the attacker's input for username is

```
$username = "'; SHUTDOWN; --";
```

This injection causes the stored procedure to generate the following query:

```
$query = "SELECT* FROM USER
WHERE username= ''; SHUTDOWN; --' AND password=''";
```

At this point, it works like a piggybacked SQLIA which separates multiple queries by semicolon. In this example, the first query is executed then, at the same time the second query is executed which causes DBMS to shut down.

Inference Based Attacks: The attacker injects SQL queries in such a way that regarding to logical answers database behave differently. The inference based attacks can be divided into 2 categories, BLIND injection and TIMING injection.

BLIND Injection Attacks [12]: Sometimes, developers hide error message details which help malicious users to make attacks to the database. In this situation attacker, instead of an error message face to a generic page provided by developer [6]. Therefore, making SQLIAs would be more difficult.

By this approach, attacker can still steal data by asking true/false questions via injected malicious SQL query, meaning that the attacker does not need to see any error messages in order to run his/her attack on the database.

For example, let's say we have an example web site that has different profiles and each user has an ID number assigned to each user to identify their profile.

Let's say ID=2000, belongs to userA. When the below URL is loaded, it is going to display the user's details which are retrieved from a database such as name, date of birth, profile photo etc.

```
http:www.examplesocialwebsite.com?ID=2000
```

The SQL statement used for this request is

```
SELECT name, description, profilePhoto, DOB FROM
USERS
WHERE id = 2000
```

The attacker may change the request to the following:

```
http:www.examplesocialwebsite.com?ID=2000 AND 1=2
```

The SQL statement changes to

```
SELECT name, description, profilePhoto, DOB FROM
USERS
WHERE id = 2000 AND 1=2
```

This will cause the query to return false, and 'Page not found' will be displayed. The attacker then proceeds to change the request to

```
http:www.examplesocialwebsite.com?ID=2000 AND 1=1
```

And the SQL statement changes to

```
$query = "SELECT name, description, profilePhoto, DOB
FROM USERS WHERE id = 2000 AND 1=1"
```

If the application is secured, both queries would be unsuccessful, because of input validation. However, if the second one returns true, meaning that the details of user with ID 2000 are shown, then this is clear that the page is vulnerable SQLIA.

TIMING Injection Attacks [4]: In this type of attacks, attacker gathers information based on response time delays in the database's responses. This attack is similar to blind injection technique and attacker can then measure the time the page takes to load to determine if the injected statement is true [6].

This technique uses an if-then statement for injecting queries. Note that, WAITFOR and SLEEP are keywords along the branches, which causes the database to delay its response by a specified time [4].

Considering the previous example, first the attacker

measures how long it takes for the web server to respond to a normal query. Then attacker issues the following request:
http:www.examplesocialwebsite.com?ID=2000 AND if(1=1, sleep(10), false)

Since the 1=1 is always true, the database will pause by 10 seconds, and it indicates that the web application is vulnerable to timing attacks.

Alternate Encodings [13]: In this type of attacks, attackers change the injected query by using alternate encoding, such as hexadecimal, ASCII, and Unicode. Because by this way they can escape from developer's filter which scan input queries for special known "bad character". Lets again consider the login example, if the following input is inserted into the username field.

$username = "0; exec (0x73587574 64 5f776e)--";

Then the query is going to be as follows:
$query="SELECT * FROM USERS
WHERE username=0; exec (0x73587574 64 5f77 6e)-- AND password="";

In this example, char () function takes hexadecimal encoding of characters and converts it into actual characters. This encoded string is translated into SHUTDOWN command which causes database to shut down.

V. Detection and Prevention Techniques for SQL Injection

A proper defense mechanism should exhibit the following properties [4].

Detection: The defense system should be able to detect and identify an SQL injection attempt.

Prevention: The defense system should have perfect knowledge of SQL injection vulnerabilities and be able pinpoint such vulnerabilities within the application.

Many frameworks in the literature have been used and/or proposed to detect and prevent SQL injection vulnerabilities in Web applications. We now list some of the most notable ones.

A. Pattern Matching Algorithm [14]

In this approach, Aho-Corasick pattern matching algorithm is used to detect and prevent SQLIA. The algorithm has two phases: static and dynamic phase:

In static phase, user generated SQL queries are compared with the static pattern list which has sample of well known attack patterns [14]. If the generated SQL exactly match with one of the patterns given in static pattern list, then it means there is an attempt to SQLIA. Otherwise, anomaly score value of the pattern which is high matching score in static pattern list will be calculated for this query in dynamic phase by Aho Corasick algorithm. If the anomaly score value is more than a given threshold value, and an alarm returns to the administrator. As soon as the alarm received by administrator, the query will be analyzed manually. If administrator detects an SQLI attack attempt, then the query will be rejected and static pattern list will be updated by this malicious pattern.

B. SQLRand [15]

The basic idea behind SQLRand is randomizing SQL commands in which the template query inside the application will be randomized. In this way SQL commands which are injected by a malicious user will not be encoded, therefore the proxy will not recognize the injected commands and the attack will not be successful.

In SQLRand, proxy server is used between web server and database server. In this technique, SQL keywords are modified by appending a random integer which is not easy to be guessed by the attacker. Before SQL statements enter the database, the modified SQL keywords are decoded into original SQL commands by the proxy. In this approach, random integers will not be appended to any other SQL commands (which are injected by the malicious user), therefore the proxy will not recognize these injected commands and lead to invalid expressions [15].

For filtering the database error messages, the proxies can be used. Proxy hides the error messages which are generated by database because of malicious queries. Remember that, error messages help the malicious user (attacker) to gather data from database table schema.

C. Query Tokenization Method [16]

This method consists of tokenizing the original query and the query with injection. The tokenization is done for both original and injected query. These tokens form an array. The lengths of arrays obtained from original query and query with injection are compared if there is a match there is not an SQL injection, otherwise there is an attempt to SQLIA.

Fig. 1 is an example for this approach.

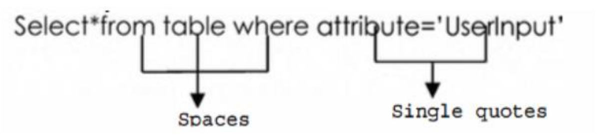
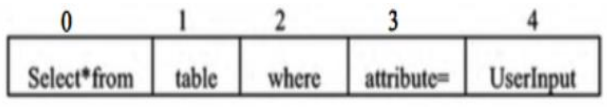


Fig. 1. Query tokenization [16].

In the above query, if Query Tokenization method is applied for detecting SQLIA, the tokens are going to be as follows:

Token0="SELECT*FROM", Token1= "Table", Token2= "WHERE", Token3= "attribute=", Token4= "Input". The tokenization is done by detecting a space, single quote or double dashes and all strings before each symbol constitute a token [16]. Therefore, the corresponding array of tokens will be as Fig. 2.



0	1	2	3	4
Select*from	table	where	attribute=	UserInput

Fig. 2. Array of tokens [16].

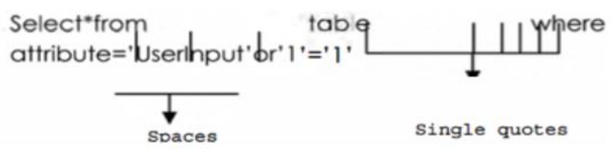
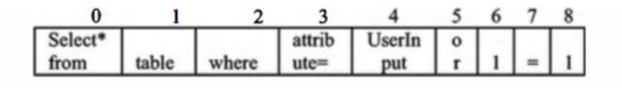


Fig. 3. Token detection [16].

By the same approach, tokenization is done for Fig. 3 as shown in Fig. 4.



0	1	2	3	4	5	6	7	8
Select* from	table	where	attrib ute=	UserIn put	o r	1	=	1

Fig. 4. Tokenization applied to injected query [16].

The main idea of this method is detection of SQLIA, by comparing lengths of the resulting arrays from the two queries. The array length of original query, as shown in the figure is 5, and the length of the injected query is 9. Therefore, since the length of the arrays are different, according to Query Tokenization method, there is an attempt to SQLIA.

D. Parse Tree Validation Approach [17]

A parse tree is a data structure which is the parsed representation of a statement. Parsing a statement requires knowledge of the language grammar that the statement was written in. Therefore, when an attacker injects a malicious SQL query as an input, then the parse tree of the original query and query with injection do not match. In this technique, parse tree of particular statement and its original statement is compared at run time. The execution of statement is stopped unless there is a match [17]. Here is an example for this approach. Fig. 5 indicates the parse tree of original query for login system.
$query = "SELECT *FROM usertable
WHERE username ='greg' AND password='secret'";

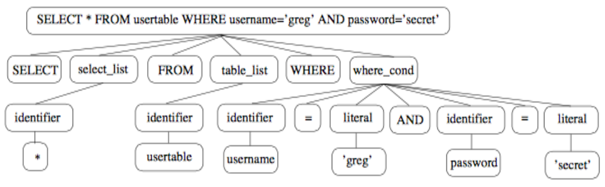


Fig. 5. Parse Tree for original query [17].

While identifiers indicate table names and attributes, literals indicate strings, numbers etc.. The following figure has a tautology based attack attempt. The malicious SQL is:
$query = "SELECT* FROM usertable
WHERE username ='greg' AND password='secret'--AND password='tricky'";

In Fig. 6, the parse tree of the original query and injected query are different. In the injected query, one more comment node is displayed.

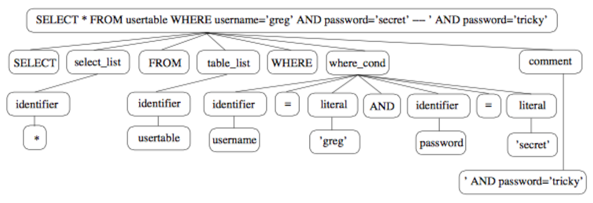


Fig. 6. Parse tree for malicious(injected) query [17].

Although, this method has strengths, it has also weaknesses: adding overhead computation and listing of inputs (black list or white list).

E. Manual Approach [18]

This approach is used for detection and prevention of SQLI vulnerabilities. It can be used in two ways.

Defensive Programming: In this way, developer implements the code in such a way that user's input can not contain any malicious SQL commands. Developers are doing this by using black and white lists. A blacklist is a basic access control mechanism that allows everyone access, except for the members of the black list (i.e. list of denied accesses). A white list is a list or register of entities that, for some reason, are being provided a particular privilege, access etc. SQL DOM, Safe Query Objects, PreparedStatement in the JDBC API, and special APIs provided by DBMSs are in this category.

Code Review: This approach is a SQL Injection detection technique with low cost but time consuming [19].

VI. Conclusion

In this paper, we have reviewed the survey of most popular SQL Injection attacks (SQLIA), vulnerabilities, detection, and prevention techniques for SQLIA.

References

[1] J. Williams, *Open Web Application Security Project. Top Ten most Critical Web Application Vulnerabilities*, 2016.
[2] A. P. A. M. Kaurl, "Token sequencing approach to prevent Sql injection attacks," *IOSR Journal of Computer Engineering*, vol. 1, no. 21-37, 2012.
[3] W. G. Halfond, J. Viegas, and A. Orso, "A classification of sql-injection attacks and countermeasures," in *Proc. the IEEE International Symposium on Secure Software Engineering*, 2006, vol. 1, pp. 13–15.
[4] A. Tajpour, M. Z. Heydari, M. Masrom, and S. Ibrahim, "Sql injection detection and prevention tools assessment," in *Proc. 2010 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT)*, 2010, vol. 9, pp. 518–522.
[5] Z. Djuric, "A black-box testing tool for detecting Sql injection vulnerabilities," in *Proc. 2013 Second International Conference on Informatics and Applications (ICIA)*, 2013, pp. 216–221.
[6] A. Tajpour, M. Massrum, and M. Z. Heydari, "Comparison of Sql injection detection and prevention techniques," in *Proc. 2010 2nd International Conference on Education Technology and Computer (ICETC)*, 2010, vol. 5, pp. V5–174.
[7] M. Medhane, "Efficient solution for Sql injection attack detection and prevention," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 3, pp. 396–398, 2013.
[8] S. McDonald, "Sql injection: Modes of attack, defense, and why it matters," *White Paper, Government Security*, 2002.
[9] D. Kar and S. Panigrahi, "Prevention of Sql injection attack using query transformation and hashing," in *Proc. 2013 IEEE 3rd International on Advance Computing Conference (IACC)*, 2013, pp. 1317–1323.
[10] M. Howard and D. LeBlanc, *Writing secure code, Pearson Education*, 2003.
[11] P. Y. Jane and M. Chaudhari, "Sqlia: Attack s by Sql injection attack and their detection mechanism," *International Journal of Engineering Research and Technology*, vol. 2, 2013.
[12] R. A. McClure and I. H. Kruger, "Sql dom: Compile time checking of dynamic sql statements," in *Proc. 27th International Conference on Software Engineering, 2005*, 2005, pp. 88–96.
[13] A. Tajpour and M. J. zade Shooshtari, "Evaluation of Sql injection detection and prevention techniques," in *Proc. 2010 Second International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN)*, 2010, pp. 216–221.
[14] M. A. Prabakar, M. Karthikeyan, and K. Marimuthu, "An efficient technique for preventing Sql injection attack using pattern matching algorithm," *Emerging Trends in Computing, Communication and Nanotechnology (ICE-CCN).*
[15] S. W. Boyd and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks," in *Proc. International Conference on Applied Cryptography and Network Security*, Springer, 2004, pp. 292–302.
[16] L. Ntagwabira and S. L. Kang, "Use of query tokenization to detect and prevent sql injection attacks," in *Proc. 2010 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT)*, 2010, vol. 2, pp. 438–440.

[17] G. Buehrer, B. W. Weide, and P. A. Sivilotti, “Usingparsetreevalidation to prevent Sql injection attacks,” in *Proc. of the 5th international workshop on Software engineering and middleware*, 2005, pp. 106–113.
[18] M. Junjin, “An approach for Sql injection vulnerability detection,” in *Proc. Sixth International Conference on Information Technology: New Generations, 2009. ITNG’09,* 2009, pp. 1411–1414.
[19] R. A. BakerJr, “Codereviewsenhancesoftwarequality,” in *Proc. of the* 19th *international conference on Software engineering*, 1997, pp. 570–571.

Gülsüm Yiğit received the BS degree in computer engineering from Zirve University, Gaziantep, Turkey, in 2014. She is currently a master student in Electronics and Computer Engineering Department from Gaziantep University. Her research interests include cloud security, secure multiparty computation.

Merve Arnavutoğlu completed her undergraduate degree at the Zirve University, Gaziantep, Turkey, in 2014. She is currently doing her master degree at Gaziantep University in Electronics and Computer Engineering Department. Her research areas are the privacy, security of iBeacon technology.

Image Processing Techniques and Methods