Understanding SQL Injection Attack Techniques and Implementation of Various Methods for Attack Detection and Prevention

Shruti Gangan[1] Tina Gyanchandani[2] Dhanamma Jagli[3]
[1,2,3]Department of Master of Computer Application
[1,2,3]VES Institute of Technology, Chembur

Abstract— Security issues of different database driven web applications are continue to be an important and crucial aspect of the ongoing development of the Internet. In the last several decades, Web applications have brought new classes of computer security vulnerabilities, such as AQL injection.
SQL Injection Attacks (SQLIAs) is one of the most severe threats to the security of database driven web applications as it compromises integrity and confidentiality of information in database. In this type of attack, an attacker gain control over the database of an application and consequently, he/she may be able to alter data. In this paper we present different types of SQL injection attacks and also implementation of different types of tools which can be used to detect and prevent these attacks.

Key words: SQL Injection, Website Security, SQL Injection Detection, SQL Injection Prevention, Sanitization

I. INTRODUCTION

SQL injection is a code injection technique in which malicious SQL statements are inserted into an entry field for execution. It is used to attack data driven applications (e.g. to dump the database contents to the attacker). It exploits security vulnerability in an application's software. SQL injection is used to attack any type of SQL database of the Websites and not on the Web Application.

A. *Examples:*

- Computer criminal defaced the Microsoft UK website using SQL injection.
- A hacker group was reported to have stolen 450,000 login credentials from Yahoo!
- CNN Website became Vulnerable.
- Sony BMG, Sony Music, Sony Pictures was breached.

1) *Format :*

Normal SQL:

SELECT * FROM users WHERE login='admin' AND pass='admin'

SQL INJECTION:

SELECT * FROM users WHERE name = ' ' OR '1'='1';

II. TYPES OF TECHNICAL IMPLEMENTATION

There are 3 types in this and they are as follows:

A. *Incorrectly Filtered Escape Characters:*

This form of SQL injection occurs when user input is not filtered for escape characters and is then passed into a SQL statement. Eg.

Expected: SELECT * FROM users WHERE name = 'username';

Injection: SELECT * FROM users WHERE name = ' ' OR '1'='1';

B. *Incorrect Type Handling:*

This form of SQL injection occurs when a user-supplied field is not strongly typed or is not checked for type constraints. Eg.

Expected: SELECT * FROM userinfo WHERE id = a_variable;

Injection: SELECT * FROM userinfo WHERE id=1; DROP TABLE users;

C. *Conditional Responses:*

One type of blind SQL injection forces the database to evaluate a logical statement on an ordinary application screen. The results of the injection are not visible to the attacker.

So the http://books.example.com/showReview.php?ID=5 would cause the server to run the query.

SELECT * FROM bookreviews WHERE ID = '5';

A hacker can load the URLs

http://books.example.com/showReview.php?ID=5 AND 1=2, which may result in queries. SELECT * FROM bookreviews WHERE ID = '5' AND '1'='2';

A blank or error page is returned from the "1=2" URL. The hacker may proceed with this query string designed to reveal the version number of MySql running on the server:

http://books.example.com/showReview.php?ID=5 AND substring(@@version,1,1)=4

The hacker can continue to use code within query strings to glean more information from the server until another avenue of attack is discovered or his or her goals are achieved.

III. ATTACK TECHNIQUES

A. *Tautology:*

This technique relies on injecting statements that are always true so that queries always return results upon evaluation of a WHERE conditional. A common example would to be inject is "or 1=1" into the "login" parameter.

Original Query: Select * from accounts where loginid="abc" and pwd="xxxxx"

Injected Query: Select * from accounts where loginid=" " or 1=1 and pwd="XXX"

B. *End of Line Comment:*

After injecting code into a particular field, usage of end of line comments. An example would be to add"- -" after inputs so that remaining queries is comments.

Original Query: Select * from accounts where loginid="abc" and pwd="xxxxx"

Injected Query: Select * from accounts where loginid=" " or 1=1 --- and pwd="XXX"

C. *Illegal/Logically Incorrect Query:*

This technique is usually used by the threat agent during the information gathering stage of the attack. Through injecting illegal/logically incorrect requests, an attacker may gain knowledge that aids the attack, such as finding out inject able parameters, data types of columns within the tables, names of tables, etc. Example - Consider the stored procedure below:

CREATE PROCEDURE sales (tot OUT number,dt in date) IS
line CONSTANT VARCHAR2(4000) :='select sum(price) from orders where odate+30>"'||dt||"'';
BEGIN
DBMS_OUTPUT.PUT_LINE('line: ' || line);
EXECUTE IMMEDIATE line into tot;
DBMS_OUTPUT.PUT_LINE('total sales:'||tot);
END;

The attacker inputs date as „5a, which is an incorrect date format. The error displayed is javax.servlet.ServletException: ORA-06550: line 1, column 7: PLS-00306: wrong number or types of arguments in call to 'SALES' ORA-06550: line 1, column 7: PL/SQL: Statement ignore….

Two important pieces of information can be inferred from the above error. First, the name of the procedure is SALES and second, the name of the database is Oracle.

D. *Union Query:*

Threat agents use this technique to guide servers to return data that were not intended to be returned by the developers. A common example would be to add the statement "UNION SELECT", along with an additional target dataset so that queries return the union of the intended dataset with the target dataset.

Original Query- 'Select salary from employee where empid='1234'‟

Injected Query- 'Select salary from employee where empid=' ' Union select * from employee'

E. *Piggy-Backed Query:*

Add additional queries beyond the intended query, effectively"piggy-backing". It is used to inject a new SQL query to the original SQL query, through the front-end of an application using query delimiter.

Original Query- „Select * from employee where empid=‟1234‟ and password=‟xxxx‟

Injected Query- Select * from employee where empid=‟ ‟; Delete from employee where empid= 1234 --' and password=‟not required‟‟‟

F. *System Stored Procedure:*

Database server often ship with system stored procedures that programmers may use when developing application. Stored procedures may yield results that go beyond the database itself, but also interact with the OS, for example, The attacker injects the code [‘';SHUTDOWN;--] in either of the two fields.

G. *Alternate Encodings:*

The attacker uses Alternate Encodings like, ASCII, to inject code. Original Query- Select * from login where username = ‘a123' and pwd=‟xxx‟

Injected Query- Select * from login where username = ‘'exec(char(0x73687574646f776e)) --‟ and pwd=‟not required‟

Result- The value passed to the char() function is the hexadecimal encoding for SHUTDOWN. Thus will cause the SHUTDOWN command to be executed.

IV. DETECTION AND PREVENTION TOOLS

Various tools used to detect and prevent attacks are:

A. *CANDID:*

It is a tool developed to guard Web applications in Java language against SQL Injection attacks. It uses candidate inputs to dynamically infer about the programmer intended query structure. Candid consists of two components: an offline Java program transformer and an online SQL parse tree checker.

B. *AMNESIA:*

Detection and prevention technique, which uses static and dynamic analysis in combination. During static analysis, it predicts the legitimate queries that can be generated by the application. During dynamic analysis, it uses runtime monitoring to check the queries generated in static analysis against the actual set of generated queries.

C. *Positive Tainting:*

Dynamic approach to detect and prevent SQL injections by performing dynamic tainting. Firstly, it finds and highlights the trusted data. Then it performs accurate taint propagation by highlighting the trusted data at character level. Finally, it performs syntax-aware evaluation of the queries, where all queries containing entrusted characters are blocked from execution on the database.

D. *SQL DOM:*

Object oriented model in which SQL queries are generated by objects which are strongly-typed to the database. It inspects the dynamically generated queries at of compile time.

E. *JDBC Checker:*

It is a static checking technique which checks for the correctness of the dynamically-generated SQL queries.

SQL Rand: The concept of Instruction-Set randomization is applied to the SQL language to detect and abort queries which contain injected code. Here, each SQL keyword is joined with a random integer to mislead the attacker.

F. *Viper:*

A tool used for Web Application penetration testing which uses heuristic approach for detecting SQL Injections.

G. *SQL-Prob:*

SQL Proxy-based Blocker which fetches the user input from the SQL query of the application and checks it against the syntactic structure of the query.

1) ADMIRE: It is a threat risk model which provides a thorough and step-by-step technique to identify and moderate the effect of SQL Injections.
2) WAVES: A Black box technique which searches for vulnerable locations in a Web application using a Web Crawler and then builds attacks which target

these locations. Finally, it watches the responses of the Web application to these attacks using machine learning techniques.

V. AMNESIA

It is Analysis and Monitoring for Neutralizing SQL-Injection Attacks, Detection and prevention technique, which uses static and dynamic analysis in combination. During static analysis, it predicts the legitimate queries that can be generated by the application. During dynamic analysis, it uses runtime monitoring to check the queries generated in static analysis against the actual set of generated queries. AMNESIA is generally combining static analysis and runtime monitoring. Static program analysis is the analysis of computer software that is performed without actually executing programs (analysis performed on executing programs is known as dynamic analysis).

AMNESIA consists of four steps:

A. Identifying hotspots:

This step performs a simple scanning of the application code to identify hotspots.

B. Building SQL-query models:

For each hotspot, build a model that represents all the possible SQL queries that may be generated at that hotspot. It uses Java String Analysis (JSA).The JSA library produces a non-deterministic finite automaton (NDFA) that expresses, at the character level, all the possible values the considered string can assume. The string analysis is conservative, so the NDFA for a string is an overestimate of all the possible values of the string. To produce the final SQL-query model, we perform an analysis of the NDFA and transform it into a model in which all of the transitions represent semantically meaningful tokens in the SQL language. This operation creates an NDFA in which all of the transitions are annotated with SQL keywords, operators, or literal values. In our model, we mark transitions that correspond to externally defined strings with the symbol beta.

To illustrate, Figure 1 shows the SQL-query model for the hotspot in the example. The model reflects the two different query strings that can be generated by the code. In the model, beta marks the position of the user-supplied inputs in the query string.

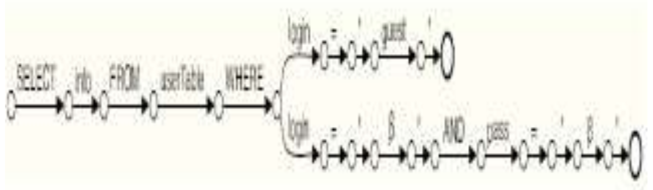
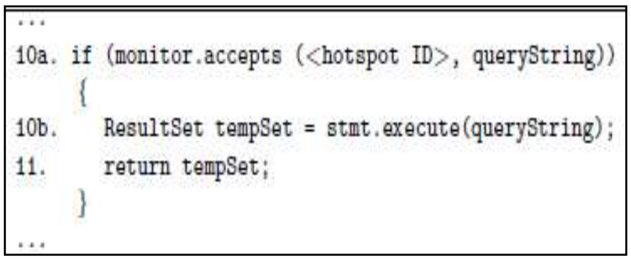


Fig. 1: SQL-query model for the hotspot

C. Instrument Application:

In this step, we instrument the application code with calls to a monitor that checks the queries at runtime. For each hotspot, we insert a call to the monitor before the call to the database. The monitor is invoked with two parameters: the query string that is about to be submitted and a unique identifier for the hotspot. The monitor uses the identifier to retrieve the SQL-query model for that hotspot. The hotspot, is now guarded by a call to the monitor at line 10a in figure 2.



```
...
10a. if (monitor.accepts (<hotspot ID>, queryString))
     {
10b.    ResultSet tempSet = stmt.execute(queryString);
11.     return tempSet;
     }
...
```

Fig. 2: hotspot, is now guarded by a call to the monitor at line 10a

D. Runtime monitoring:

At runtime, the application executes normally until it reaches a hotspot. At this point, the query string is sent to the runtime monitor. The monitor parses the query string into a sequence of tokens according to the specific SQL dialect considered. Figure 3. shows how the last two queries discussed would be parsed during runtime monitoring. After parsing the query, the runtime monitor checks whether the query violates the hotspot's SQL-query model. To do this, the runtime monitor checks whether the model accepts the sequence of tokens in the query string. When matching the query string against the SQL-query model, a token that corresponds to a numeric or string constant (including the empty string, '') can match either an identical literal value or a beta label. If the model does not accept the sequence of tokens, the monitor identifies the query as an SQLIA.

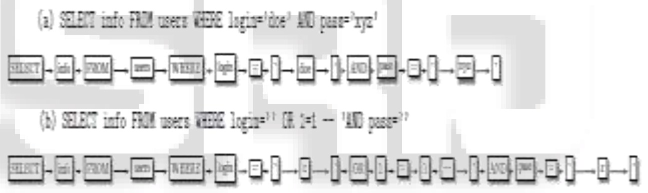

(a) SELECT info FROM users WHERE login='doe' AND pass='xyz'
(b) SELECT info FROM users WHERE login='' OR 1=1 -- 'AND pass=''

Fig. 3: Queries Parsed During Runtime Monitoring

VI. AMNESIA IMPLEMENTATION

Implementation consists of three modules:

1) Analysis module:

It implements Steps 1 & 2. It inputs JSP pages and outputs a list of hotspots. It then builds SQL-query models for each hotspot.

2) Instrumentation module:

It implements Step 3. It instruments each hotspot with a call to the runtime monitor.

3) Runtime-monitoring module:

It implements Step 4. It inputs a query string and the hotspot ID. It retrieves the SQL-query model for that hotspot and then matches SQL-query model with the submitted query string.

VII. PREVENTION USING WEB SERVICE

The technique consists of two filtration models to prevent SQLIA'S

1) Active Guard filtration model
2) Service Detector filtration model
3) Web Service Layer

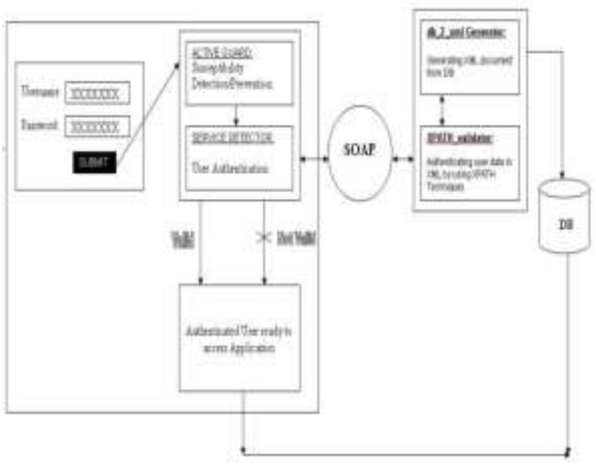


Fig. 4: Web Service System Architecture

The steps are summarized and then describe them in more detail in following sections.

A. Active Guard Filtration Model

Active Guard Filtration Model in application layer build a Susceptibility detector to detect and prevent the Susceptibility characters or Meta characters to prevent the malicious attacks from accessing the data's from database.

B. Service Detector Filtration Model

Service Detector Filtration Model in application layer validates user input from XPATH_Validator where the Sensitive data's are stored from the Database at second the data existed in XPATH_Validator if it is identical then the Authenticated /legitimate user is allowed to proceed.

C. Web Service Layer

Web service builds two types of execution process that are DB_2_Xml generator and XPATH_ Validator. DB_2_Xml generator is used to create a separate temporary storage of Xml document from database where the Sensitive data's are stored in XPATH_ Validator, The user input field from the Service Detector compare with the data existed in XPATH_ Validator, if the data's are similar XPATH_ Validator send a flag with the count iterator value = 1 to the Service Detector by signifyingthe user data is valid.

D. Identify hotspot

This step performs a simple scanning of the application code to identify hotspots. Each hotspot will be verified with the Active Server to remove the susceptibility character the sample code.(In .NET based applications, interactions with the database occur through calls to specific methods in the System. Data.Sqlclient namespace, such as Sqlcommand- . ExecuteReader (String)) the hotspot is instrumented with monitor code, which matches dynamically generated queries against query models. If a generated query is matched with Active Guard, then it is considered an attack.

VIII. SQL PROB

SQLProb(SQL Proxy-based Blocker) extracts user input from the application-generated query. The user input data has been embedded into the query, and validate them in the context of the generated query's syntactic structure by using Validation algorithm. It is a complete black-box approach that does not require modifying application or database code. Input validation technique does not require metadata or learning. It utilizes an off-the-shelf proxy that requires minimal setup complexity. It is independent of the programming language used in the web application.

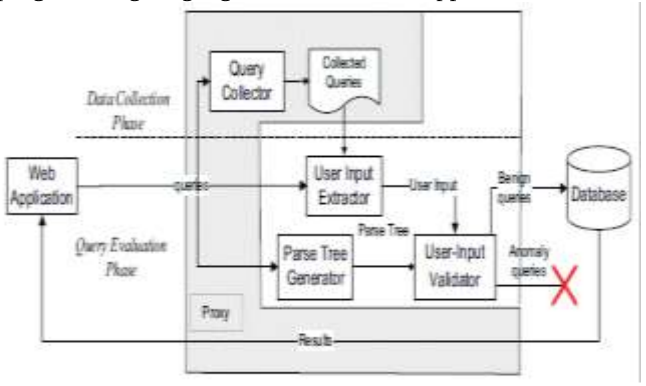


Fig. 5: SQLProb Architecture

A. SQLProb has four main components:

1) The Query Collector processes all possible SQL queries during the data collection phase.
2) The User Input Extractor implements a global pairwise alignment algorithm to identify user input data.
3) The Parse Tree Generator generates the parse tree for the incoming queries.
4) The User Input Validator evaluates whether the user input is normal or malicious based on user input validation algorithms.

1) Input Extraction

The algorithm uses four steps:

1) scoring similarity
2) summing,
3) back-tracking
4) finalization

2) Parse Tree

A concrete syntax tree or parse tree derivation tree is an ordered, rooted tree. It represents the syntactic structure of a string according to some context-free grammar.

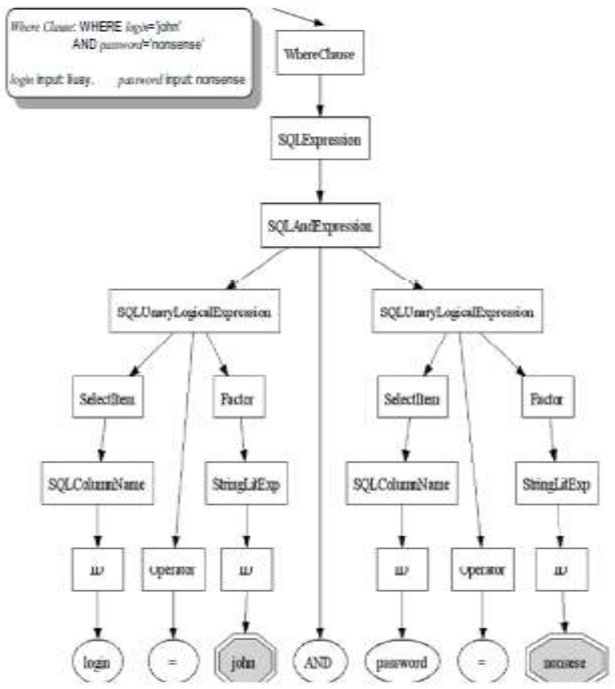


Fig. 6: Parse Tree

3) Validation Algorithm

```
Data: Parse Tree $Tree(q_i)$ and the set of user input $UI(q_i)$
Result: True if the query is an SQLIA, or False if otherwise
1. for every user input $UI_{i,j}$ in $UI(q_i)$
    2. do depth-first-search upward from every leaf node $leaf(u_i)$ parsed from $UI_{i,j}$, according to SQL grammar $\mathcal{G}$;
    3. Searching stops when all the searching path intersect at a non-leaf node nl_node;
    4. do breath-first-search downward from nl_node until reaching all $m$ leaf nodes $leaf(node)_k$;
    5. if $\bigcup_{i=1}^{n}(leaf(u_i)) \subset \bigcup_{k=1}^{m}(leaf(node)_k)$ then
        Return True;
        else Return False;
    end
end
```

B. SQL Prob Implementation

1) Input Validation

Because the root cause of SQLIAs is the intermingling of data and control code, improper input validation accounts for most security problems in database and web applications.

2) Static Analysis

To guarantee security, perform static analysis over the entire application's source code to ensure that every piece of input is subject to an input validation check before being incorporated to a query.

3) Learning-based Prevention

A set of learning-based approaches have been proposed to learn all the intended query structure statically or dynamically.

4) Dynamic Prevention

Dynamic tainting approaches taint the input strings and track those taints along the information flow of a program.

IX. CONCLUSION

In this paper, we have tried to explain how to detect and prevent the SQL Injection attack on the websites. The different methods and prevention techniques presented here will be beneficial for security of a website. But obviously, hackers are always active in discovering the new techniques for attacking web information. Therefore, further investigation in this domain will be required to find new approaches and protection measures to cope with them.

ACKNOWLEDGMENT

We have immense happiness in expressing our sincere thanks to all our faculty members whose guidance always inspire us to do better. We would like to thank all the people who helped us directly or indirectly. We are also thankful to our parents and friends for their motivated support in completion of this work.

REFERENCES

[1] Kirti Randhe,Vishal Mogal,"Defense against SQL Injection and Cross Site Scripting Vulnerabilities", International Journal of Science and Research (IJSR) Volume 3 Issue 11, November 2014 ISSN (Online): 2319-7064

[2] Priyanka, Vijay Kumar Bohat, "Detection of SQL Injection Attack and Various Prevention Strategies" International Journal of Engineering and Advanced Technology (IJEAT) Volume-2, Issue-4, April 2013 ISSN: 2249 – 8958

[3] Indrani Balasundaram, Dr. E. Ramaraj, "An Approach to Detect and Prevent SQL Injection Attacks in Database Using Web Service,"International Journal of Computer Science and Network Security (IJCSNS), VOL.11 No.1, January 2011.

[4] William G.J. Halfond and Alessandro Orso "Preventing SQL Injection Attacks Using AMNESIA", ICSE'06, May 20–28, 2006, Shanghai, China.

[5] Anyi Liu, Yi Yuan,Duminda Wijesekera, "SQLProb: A Proxy-based Architecture towards Preventing SQL Injection Attacks", SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

[6] http://en.wikipedia.org/wiki/SQL_injection

[7] https://www.acunetix.com/websitesecurity/sql-injection/