SQL Injections – A threat to Web Applications

Juhi Gupta[1] | Ruchi Singhal[2]

[1] (Assistant Professor, Department of Computer Science, Maharaja Agrasen College, University of Delhi, India, juhiaqua26@gmail.com)

[2] (System Engineer, Tata Consultancy Services,Gurgaon, India , ruchisinghal07@gmail.com)

***Abstract*:** *SQL injections have become a serious threat to the integrity, confidentiality and security of web applications. Attackers can gain unauthorized access to the database and can cause serious damage to the web application. Researchers have proposed various solutions to this problem. Many tools have also been devised to deal with this problem but each come with a limitation. In this paper, study about SQL injections has been done. Various types of SQL injection and tools to counter them has been discussed in this paper. For each technique, we have discussed its strengths and weaknesses in addressing the entire range of SQL injection attacks.*

Keywords: SQL Injection Attacks (SQLIA); query; database; web application

1. INTRODUCTION

Today in the era of internet, web applications are most vulnerable to attacks by hackers. Web applications are being developed by many organizations to provide services to their users. They receive inputs from the users, interact with their underlying databases and then return the relevant response. The confidential and sensitive information contained in the back end database interest attackers. SQL Injection is the hacking technique, used to attack data driven applications that take advantage of lack of input validation. In this technique an attacker attempts to create or alters SQL commands (statements) for execution by the backend database in order to expose hidden data.

SQL injection attacks are the common threat to the security and integrity of web applications. An SQL injection attack can successfully modify the data in database (Insert/Delete/Update), read confidential and sensitive data from it, execute various administrative operations such as shutdown the DBMS and in some cases may even issue commands to operating system. It means that SQL queries are thereby bypassing standard authentication and authorization checks i.e. they are able to circumvent access controls.Irrespective of sufficient network security equipment and all the intrusion detection system installed before the physical database server, a hacker will have clear channel (or tunnel) of communication to the database.

Insufficient validation of user input is the main cause of SQL injection vulnerabilities. To address this problem, a range of coding guidelines have been proposed by many developers that promote defensive coding practices, such as encoding user input and validation[1]. A rigorous and systematic application of these techniques is an effective solution for preventing SQL injection vulnerabilities. However, in practice, the application of such techniques is human-based and, thus, prone to errors. Furthermore, fixing legacy code-bases that might contain SQL injection vulnerabilities can be an extremely labor-intensive task.

Firewalls and similar intrusion detection mechanisms are not capable of providing full defense against SQL Injection web attacks. Several techniques have been proposed to prevent SQL injection attacks. This paper surveys various prevention techniques and detection tools for SQL injections.

2. TYPES OF SQL INJECTION ATTACKS

There are various types of SQL injection attacks that can be performed together or sequentially on web applications depending on the intent of attacker. In this paper different kinds of SQL injection attacks are discussed along with the example. Before discussing the different types of SQL injection attacks, we present an example application written using Java servlet which is vulnerable to SQL injection.

```
1.  String empID, password, query
2.  empID= getParameter("Employee id");
3.  password = getParameter("pwd");
4.  Connection
con.createConnection("DatabaseName");
5.  query = "SELECT * FROM users WHERE
empID='" +
6.   empID + "' AND pass='" + password " ' ;
7.  Resultset result = con.executeQuery(query);
8.  If (result!=NULL)
9.   displayHomePage(result);
10.  Else
11.   displayInvalidUser();
```

The code snippet above implements a simple login functionality which is common in most of the web applications. This example is used in this paper for illustrative purposes as it is simple to understand. This code uses two input parameters i.e. employee ID and its password to build a dynamic SQL query and submit it to the database.

For example, if the user submits empID as '681563' and password as 'abc' then the application dynamically builds the following SQL query and submits it to the database

SELECT employee FROM users WHERE empID="681563" AND pass ="abc"

If the employee ID and the password matches the one stored in backend database then home page for the employee is displayed by calling displayHomePage() function otherwise error message is displayed to the unauthorized user. Now the following classification of SQLIAs [1, 2] will be presented.

2.1 TAUTOLOGIES

The tautology based SQL attack aims to inject code in conditional statements so that they always evaluate to true. This type of attack is commonly used to bypass authentication pages and extract sensitive data from the database. The attacker exploits the vulnerable field that is used in the Where condition of the query. This is also known as first order

Injection attack where the attacker simply enter a malicious string and causes the modified string to be executed immediately [3]. All the rows in the targeted table of the database are returned to the attacker by transforming the conditional statement into tautology. Example: In this example attack, the attacker enter malicious data into the input field i.e.” ‘ ‘ OR ‘1’ =’1’ in the employee id and password field. The resulting query is :

SELECT * FROM users WHERE empID = ' ' OR '1'='1' AND pass = ' ' OR '1'='1';

This code transforms the entire Where clause into tautology which is always true. The query evaluates true for every row in the backend database and returns all of them [1,4].

2.2 ILLEGAL/LOGICALLY INCORRECT QUERIES

Often, developers use inbuilt error handing libraries and functions which help in the debugging and code fixing process. These functions deliver error messages on the screen which can reveal lot of sensitive data about the application and attacker can even gain information about the schema of the database. They are also known as Error based SQL injections. In this type of attack, the attacker intentionally injects junk input which causes the syntax, type mismatch or logical errors in the database. Vulnerable or injectable parameters can be easily identified with the help of syntax errors. Data type of input fields can be deduced from type mismatch error. Logical errors often reveal the names of the tables and columns that caused the error [1].

Example: This example attack’s goal is to find out whether the input field is injectable or not by causing type mismatch error. Suppose there is a script likehttp://abcsite.com/loginscript.php?id=1and we have to find out if it is vulnerable to SQL injection. The attacker might inject the following code in the URL

SQL INJECTION:

http://abcsite.com/loginscript.php?id=1’

Error message depends on the quality of script. If the script filters it for SQL keywords then no SQL error would be returned but if the script has no filtering mechanism then the attacker might get an error like this

"MySQL Syntax Error By '1" In file loginscript.php On Line 9."

This shows that server does not filter the input fields for SQL command and is injectable.

2.3 PIGGY BACKED QUERIES

In this type of attack, the attacker’s intent is to inject additional query thereby modifying data, performing denial of service operation or execute remote commands. Here, the existing query is not modified instead an additional query is piggybacked onto the original query. The attacker exploits the database by using query delimiter (;). Web applications having database configuration that allows multiple statements in a single string are vulnerable to this type of attack.

Example: If the intruder or attacker inputs “ ‘; INSERT INTO users (empID, pass) VALUES (‘attacker employeid’,’ attacker password’);--” into the pass field, the application generates the query:

SELECT * FROM users WHERE empID=’681563’ AND pass= ‘ ‘; INSERT INTO users (empID, pass) VALUES (‘attacker employeid’,’ attacker password’);-- ’

The database would recognize the query delimiter after executing the first query and would execute the injected query. The result of the query would be to add another user in the users table which attacker can use subsequently to login into homepage.

2.4 UNION QUERY

With this technique, an attacker may retrieve the data from other tables as well by injecting SQL statement of the form UNION SELECT <injected query>. Since the attacker is in control of the second query, it can be used to extract data from unrelated tables. The UNION SELECT statement allows chaining of two queries that are unrelated i.e. having nothing in common.

Example:

SELECT * FROM users WHERE empID= ‘ ‘ UNIONSELECT * FROM AccountDetails WHERE empName =’abc’ AND pass = ‘ ‘

The first query will return the null set as there will be no row in the database where the empID field is null but the second query will return the details from AccountDetails table. The result of the query will be union of the result set of these two queries.

2.5 INFERENCE

In this type of injection, the attackers are trying to attack website which is highly secured. Even when an injection has been succeeded, no useful information or feedback is provided by the database error messages. In this situation, the attacker enters various commands into the site to trigger noticeable changes in the responses of the website. The attacker can then deduce the vulnerable parameters as well as additional information about the values in the database by carefully noting the behavior of the database. Two well known techniques that are based on inference are

- BlindInjection

In this injection, the attacker asks server true/false questions and then infers the behavior of the page depending on the answer. The word ‘blind’ comes from the fact that the injector is blindly injecting commands using some calculated assumptions and tries [6].

Example : let us consider two injections for our running example.

SELECT * FROM users WHERE empID = ‘validID’ AND 1=0 - - ‘ AND pass= ‘ ‘

SELECT * FROM users WHERE empID = ‘validID’ AND 1=1 - - ‘ AND pass= ‘ ‘

After the execution of these two injections, attacker might come across two scenarios. In the first scenario, the application is secured enough and correctly validates the empIDfield. In this case, application would return error messages upon execution of both injections and the attacker would infer that the empID field is not vulnerable. In the second scenario, the application is unsecured and is vulnerable to SQL injection. In this case, upon execution of first injection, a login error message is returned but at this point the attacker doesn’t know if this is because the application

blocked the attempt and correctly validated input or because the attack itself caused the error as it always evaluates to false. The attacker then executes the second injection, which always evaluates to true (1=1). If no error message is returned by the application then the attacker would know that the input field is injectable and the attack went through.

• Timing Attacks

It is a technique in which attacker retrieve information from the database by observing timing delays in the response of database [1]. This technique uses a different method of attack than blind injection. In this method, the attacker asks yes/no questions from the database by injecting a conditional time delay in the query. The time delay will be executed depending on if the condition is true or false and the server will take abnormally long time to respond. In this way, the attacker gets to know if the condition was true or false and therefore the answer to the injected question.

Example:

SELECT * FROM usersWHEREempID=1; IF SYSTEM_USER='sa' WAIT FOR DELAY '00:00:17'

Using this query injection, the attacker would be able to check if the user is sa (system administrator) based on the response from server.

2.6 STORED PROCEDURES

Today, many databases come with standard set of stored procedures which aim to extend the functionality of the database and also allow for interaction with the operating system. Attackers can easily determine which backend database is in use and then craft their SQL injection attacks which can execute stored procedures provided by that database. Developers are often surprised to find that their stored procedures can be just as vulnerable to attacks as their normal applications [4,7].

Example :

Stored procedure for authentication

CREATE PROCEDURE DBO.isAuthenticated@userName varchar2, @pass varchar2

AS

EXEC("SELECT * FROM users WHERE empID='" +@userName+ "' and pass='" +@password "');

GO

To launch SQL injection attack (SQLIA), the attacker simply inserts " ' ; SHUTDOWN; - -" into either inputfields. The stored procedure generates the following query:

SELECT * FROM users WHERE empID='681563' AND pass=' '; SHUTDOWN; --

This attack works as piggy back query attack, First query is executed normally then the second malicious query is executed which causes database to shutdown.

2.7 ALTERNATE ENCODINGS

This is a technique which is often used by attackers in order to avoid detection by many automated prevention techniques and defensive coding practices. In this type of attack, the attacker modifies the injected text by using alternate encodings such as ASCII, hexadecimal and Unicode. Developers often employs common defensive coding practices which scans certain known characters ("bad characters") such as comment operators and single quotes. To evade this defense, attackers have employed alternate methods of encoding injected query.

Example: In this attack, the following text is injected into the empID field:"legalUser'; exec(0x7368757464646f776e) - - ". The query that will be generated after this injection is:

SELECT * FROM users WHERE empID='legalUser';

exec(char(0x73687574646f776e)) -- AND pass=''

In this example, char() function and ASCII hexadecimal coding is used. This function takes integer parameter and then return an instance of that character. The hexadecimal character stream used in this injection corresponds to the string "SHUTDOWN". Therefore, this injection results in the execution of SHUTDOWN command by the database.

3. SQL INJECTION DETECTION AND PREVENTION TOOLS

Researchers have found that defensive coding techniques or OS hardening are not enough to stop SQLIA on web applications so many tools have developed for the task. Various tools that have been developed for prevention and detection of SQLIA are discussed in this paper.

Huang and colleagues [8] proposed WAVES,a black-box technique for testing Web applications for SQL injectionvulnerabilities. This technique identifies all vulnerable points in the web application where attacker can inject SQL injection using a web crawler. It then attacks those target points using specified list of patterns and attack techniques. WAVES then analyses the response returned by the application and improves its attack methodology. This tool does not guarantees of completeness.

Static Code Checker or JDBC- Checker can be used to prevent SQLIA which take advantage of type mismatches in the dynamically generated query string. This checker can detect one of the root cause of major SQLIA vulnerabilities in code i.e. improper type checking of input but its scope is limited as it cannot detect other types of attacks.

SQL Guard and SQL Check approaches check queries at runtime based on a model of expected queries which is expressed as a grammar that only accepts legal queries. In SQL Guard, first the structure of the query is examined before and after the addition of user input and then the model is deduced at runtime whereas in SQLCheck, the model is specified independently by the developer. In both approaches, a secret key is used which delimit user input during parsing. Thus, security depends on attacker not being able to discover the key. Also, developers rewrite code to use a special intermediate library or manually insert special markers where user input is added to a dynamically generated query [1].

AMNESIA is a model based technique which involves runtime monitoring as well as static analysis. In the static phase, AMNESIA builds models of different queries that an application can generate at each access point to the database. It then intercepts all queries before sending them to the database and checks each query against the statically built models in the dynamic phase. SQLIAs are identified as the queries which violate the model and are prevented from executing on the database.

WebSSARI [10] use static analysis to check taint flowsagainst preconditions for sensitive functions. This

analysis detects the points where preconditions have not been met and then suggest some sanitization functions or filters which can be added to the application in order to satisfy these preconditions. The limitation of approach is adequate preconditionsfor sensitive functions cannot be accurately expressed sosome filters may be omitted.

SecuriFly is a technique that was implemented for java. Queries generated by tainted input are sanitized using this technique but this approach does not help if the injection is performed into numeric fields. Identification of all sources of tainted user input in web applications is the limitation of this technique.

Valeur and colleagues [11] proposethe use of an Intrusion Detection System(IDS) to detect SQLIAs. It is based on a machine learning technique that is trained using a set of typical application queries. The technique first builds a model of all typical queries and then monitors the response of application at runtime in order to identify queries that do not match the model. The major drawback of learning based techniques is that they are dependent on the quality of training set used and hence cannot provide guarantee about their detection capabilities.

4. CONCLUSION AND FUTURE SCOPE

In this paper, we discussed and analyzed various types of SQL injections prevalent today with the help of an example application. Then we investigated various SQL injection detection and prevention tools available. Strengths and weaknesses of each has been discussed. In our future work, we will propose a common framework that would be able to measure effectiveness, efficiency of these tools in countering SQL injection attacks.

REFERENCES

[1]. William G.J. Halfond, Jeremy Viegas and Alessandro Orso, “A Classification of SQL Injection Attacks and Countermeasures,” College of Computing Georgia Institute of Technology IEEE, 2006.

[2]. Atefeh Tajpour, Suhaimi Ibrahim, Maslin Masrom, "Evaluation of SQL Injection Detection and Prevention Techniques”. International Journal of Advancements in Computing Technology, 2011, Korea.

[3].http://download.oracle.com/oll/tutorials/SQLInjection/html/lesson 1/les01_tm_attacks.htm

[4]. M. Howard and D. LeBlanc. Writing Secure Code. Microsoft Press, Redmond, Washington, second edition, 2003

[5]. S. McDonald. SQL Injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity.org, April 2002.

[6]. http://resources.infosecinstitute.com/sql-injections-introduction

[7]. C. A. Mackay. SQL Injection Attacks and Some Tips on How to Prevent Them. Technical report, The Code Project, January 2005. http://www.codeproject.com/cs/database/SqlInjectionAttacks.asp.

[8]. Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In Proceedings of the 11th International World Wide Web Conference (WWW 03), May 2003.

[9]. Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluation. Proceedings of the 14th ACM conference on Computer and communications security. ACM, Alexandria, Virginia, USA.page:12-24.

[10]. Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In Proceedings of the 12th International World Wide Web Conference (WWW 04), May 2004.

[11]. F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Vienna, Austria, July 2005.