International Journal of Control Theory and Applications

ISSN : 0974–5572

© International Science Press

Volume 9 • Number 40 • 2016

Preventing Structured Query Language (SQL) Injection Attacks in Mobile Applications

Subburaj Ramasamy[a] and Varsha V[b]

[a]*Department of Information Technology, SRM University Chennai, India*

E-mail: subburaj.r@ktr.srmuniv.ac.in, varsha_vairavel@srmuniv.edu.in

Abstract: This article presents an approach to preventing Structured Query Language (SQL) injection attacks in a database. SQL injection attacks pose a major threat to the data present in a database. Attackers enter queries with malicious code into the user input of an application. This is not normally checked, and the query is allowed to access the database. This opens up vulnerabilities in the application that uses this database to store the information. The Confidentiality, Integrity, and Availability (CIA) of data are put at risk. Therefore to prevent such kind of attacks, a program "Vaccine" has been developed that sanitizes the user input query and allows only legitimate queries to access the database. Any suspicious query that is entered will be blocked, and an error message will be displayed. This program attempts to protect the database from 13 major vulnerabilities by putting them on a blacklist. SQLite databases are used in a few websites, and mobile applications and the Vaccine seems to prevent malicious queries.

Keywords: *SQL injection; vulnerabilities; database; attacker.*

1. INTRODUCTION

An SQL injection attack is carried out by inserting an SQL query via client's input. An SQL injection, if successful, can read sensitive data from the database, modify it, execute administration operations, retrieve the contents of a file present on the DBMS server and sometimes assign commands to the operating system of the server. These malicious SQL queries are placed to provoke the predefined SQL commands to be carried out. These kinds of attacks allow hackers to spoof an individual's identity, tamper with existing data in the database, cause repudiation issues, allow complete leakage of all data on the system, make data unavailable, and become administrators of the DBMS server. SQL Injection is prevalent in applications coded using PHP and ASP as they use older functional interfaces. However J2EE and ASP.NET applications are a bit more secure hence they are less prone to regularly exploited SQL injections due to the nature of programmatic interfaces available [1].

There are different kinds of SQL injection attacks based on how much research the attacker has done on the system. The major three categories of attacks are conventional injection attack, dictionary injection attack and blind injection attack. The first attack type makes use of the error messages returned from the application to gather information about the system. The attacker forces the server to execute an illegitimate SQL statement through the application's input. If the server does not mask the error message, it will be sent to the client. The attackers will use these error messages to obtain useful information and reconstruct the SQL query based on that. Dictionary injection attack is based on a dictionary that detects important details about the database of web applications. Normally network administrators do not give importance to information security, so most database column names are named following a common convention. This allows dictionary injection attacks to succeed. Blind injection attacks send logic statements to the server and obtain related information with the response from the server. If the application returns correct results, it means the attacker has succeeded in his/her data speculation. However, if errors are shown, then data speculation process is continued till correct results are returned. Though blind injection and conventional injection appear to be similar types, they differ in the way they interpret the error messages received from the server. The conventional injection attack method uses the error message to reconstruct the SQL query while the blind injection does not depend on the error message returned. However, both concentrate on the analysis of the injection point, collect related information about the database and other related activities [2].

SQL injection attacks are also carried out by modifying the user input query as follows:

1. **Tautologies:** This attack aims at injecting code in conditional statements so that they always return true. The main purpose is to bypass authentication pages and obtain data from them. The attack is termed efficacious when the code either displays all of the records or carries out some operation on the data.

2. **Union queries:** The attacker carries a union operation between the legitimate query and his own malicious query. This returns a data-set which is the union of the results of the original first query and the injected second query.

3. **Stored Procedures:** The aim of these attacks is to execute stored procedures present in the database. Stored procedures are a set of user defined functions that can be looked up and reused anytime. SQL attacks can be formulated to execute stored procedures provided by a specific database. Sometimes it might even include procedures that interact with the operating system. Furthermore, since stored procedures are often written in special scripting languages, they may contain other types of vulnerabilities, like buffer overflows, which will allow attackers to run arbitrary code on the server or escalate their privileges [3][4].

The most commonly used techniques for prevention against SQL injection attacks are static analysis, dynamic analysis, combined static and dynamic analysis, defensive programming and machine learning techniques. The static analysis method scrutinizes the code for vulnerability without actually executing the code. The dynamic analysis method is performed automatically by analyzing the vulnerabilities while running the application. These tests are carried out using penetration testing tools. The combined static and dynamic analysis method compensates the limitations of static and dynamic methods. It is considered highly efficient against SQL injection attacks but it is very arduous. This method analyzes the application code using static analysis and builds a model of the appropriate queries that the application can generate. At run-time, the technique overlooks all dynamically-generated queries and checks them against queries from static analysis. The machine-learning method is the most frequently used method because it gives high false positives and a low detection rate [5].

Other preventive measures to prevent SQL injection attacks are :

1. User input data should be filtered and parameterized statements should be used.

2. Using professional vulnerability scanning tools

3. Implementing security right from the beginning of the software development life cycle.

4. Restricting detailed error information from being returned as this can be used by the attacker to gain more knowledge about the application [6].

In section II, the major issues of SQL injection and how databases are affected because of this will be discussed. Section III will provide a solution for solving the various issues encountered due to SQL injection attacks. Next, Section IV focuses on the results of the proposed solution that protects databases. Finally, Section V provides a conclusion and summary of the complete study on SQL injection in mobile applications.

2. ISSUES

The mobile applications that store data locally (on the phone) use the SQLite database. The reason is that SQLite does not require an external server to store data and it is light-weight.

Normally, malicious code is provided by the attacker as input to the mobile application through different ways. The deformed data is handled (similar to genuine data) by the mobile application's underlying frameworks. On processing this erroneous data, a context switch is forced and it is reinterpreted as legitimate code by the framework. In the end, this malicious code is executed by the app. Sometimes the code runs with the same scope and access permissions as the user that has unintentionally executed this code and does not do much harm. However if the code executes with privileged permissions, it leads even more greater damage potential than the previous case. There are also other kinds of SQL injection attacks called binary attacks. It involves injecting binary code into the mobile application directly [7].

SQL injection attackers can perform different kinds of attacks with different attack intents. A few of them are listed in the following page:

1. **Identifying vulnerable parameters :** The attacker keeps probing the application to discover which parameters and user input fields are vulnerable to SQL injection attacks.

2. **Performing database finger-printing :** The attacker can discover the type and version of a database the application is using to finger-print it. This will allow the hacker to carry out database specific attacks for gaining more information.

3. **Determining database schema :** To correctly isolate data from the database, the hacker needs to have knowledge of the database schema. These kinds of attacks are used to gather information.

4. **Extracting data:** This is the most common type of SQL injection attack. These type of attacks employ techniques to extract values from the database. The information at risk could be sensitive and highly desirable to the attacker.

5. **Performing Denial of Service :** The aim of these kind of attacks is to close down the database of an application. Attacks involve locking or dropping the database tables.

6. **Evading detection:** These attacks are employed to avoid auditing and detection by system protection mechanisms.

7. **Bypassing authentication:** The aim of these kinds of attacks is to allow the attacker to bypass authentication mechanisms in the database and application. It allows the attacker to assume rights and privileges associated with another application user.

8. **Executing remote commands:** This attack attempts to execute superficial commands on the database. This is achieved using stored procedures or functions that database users operate with.

9. **Performing privilege escalation :** These attacks make use of logical errors or implementation errors in the database for escalating the privileges of the hacker. Contrary to bypassing authentication attacks, these attacks target the exploitation of database user privileges [8].

There are a lot of consequences because of SQL injection attacks. A few of them are:

1. **Authorization:** Critical data stored in a vulnerable database can be altered by an SQL injection attack using an authorization privilege.
2. **Authentication:** If there is no proper control on the authentication mechanism, the attacker may be able to log-in to the system as a normal user without knowing the username or password.
3. **Confidentiality:** Normally databases contain sensitive data such as personal information and credit card numbers. Loss of confidentiality of such important data can happen with SQL injection attacks. Theft of sensitive data is usually one of the most common intentions of attackers.
4. **Integrity:** A successful SQL injection attack can not only allow the hacker to read sensitive information but also change or delete this important information [9].

Building secure software is a challenge for software developers. A survey on security perspectives reveals that insecure software holds sixty percent of a higher business threat than software that is secured. Compilers are only meant for checking errors and not security vulnerabilities. This creates the need for analysis tools which can capture such security-vulnerabilities before run-time. The whole process of software development changes when taking security into account because normal development methods do not include security measures in each and every step of the process [10].

To avoid these kinds of SQL attacks, a security mechanism should be placed between the mobile application and the local database to check the queries generated by the user application.

The Confidentiality, Integrity, and Availability (CIA) of data are put at risk whenever data is not properly secured. Regular applications are designed with functionality in mind and security is never taken into account. Therefore it is necessary to start building applications with appropriate security features.

Though a lot of tools have been developed for providing database security, there is no completely fool-proof tool that prevents against SQL injection attacks. Therefore an attempt to develop a tool that protects against major types of vulnerabilities has been proposed.

3. SOLUTION

The proposed solution is a “program Vaccine” in the Intrusion Detection System that sanitizes the user input queries. All queries passing through the application will enter this program before being allowed to access the database.

Fig. 1 shows the typical architecture of a 2 tier application.

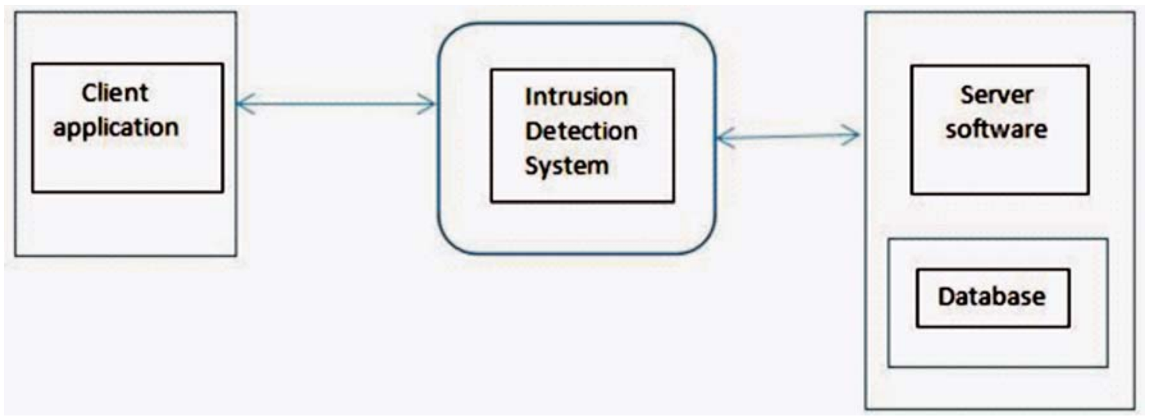


Figure 1: Architecture of a typical 2 tier application

The role of each component is explained below:

1. **Client application :** This is the component which contains the client application. It is the place where the user interacts with the application software.
2. **Intrusion Detection System:** This is the intermediate component which comprises of firewalls and other security mechanisms. The tool for checking the SQL queries "Vaccine" will also be placed here right before the server software so that even if the attacker manages to get past the firewall, the erroneous query will not be able to access the database.
3. **Server Software and Database:** This is the final component which houses both the server software and the database. Typical mobile applications will allow direct access to the database. However since the tool is present as part of the Intrusion Detection System, it checks the queries before they access the database.

The "vaccine" is a program written entirely in Python and works only with SQLite databases. Since SQLite databases are created and stored locally, they are used in mobile applications. They are also lightweight and do not require an external interface for interacting with the database. The python file for Vaccine is placed between the mobile application and the database. When the application generates a query, it is passed through Vaccine. The entered query is checked against a set of predefined rules in the blacklist. If any of those characters from the blacklist are present in the query, it is blocked from the database and an error message is displayed.

Table I lists all of the major keywords that are vulnerabilities in the SQLite queries [11][12].

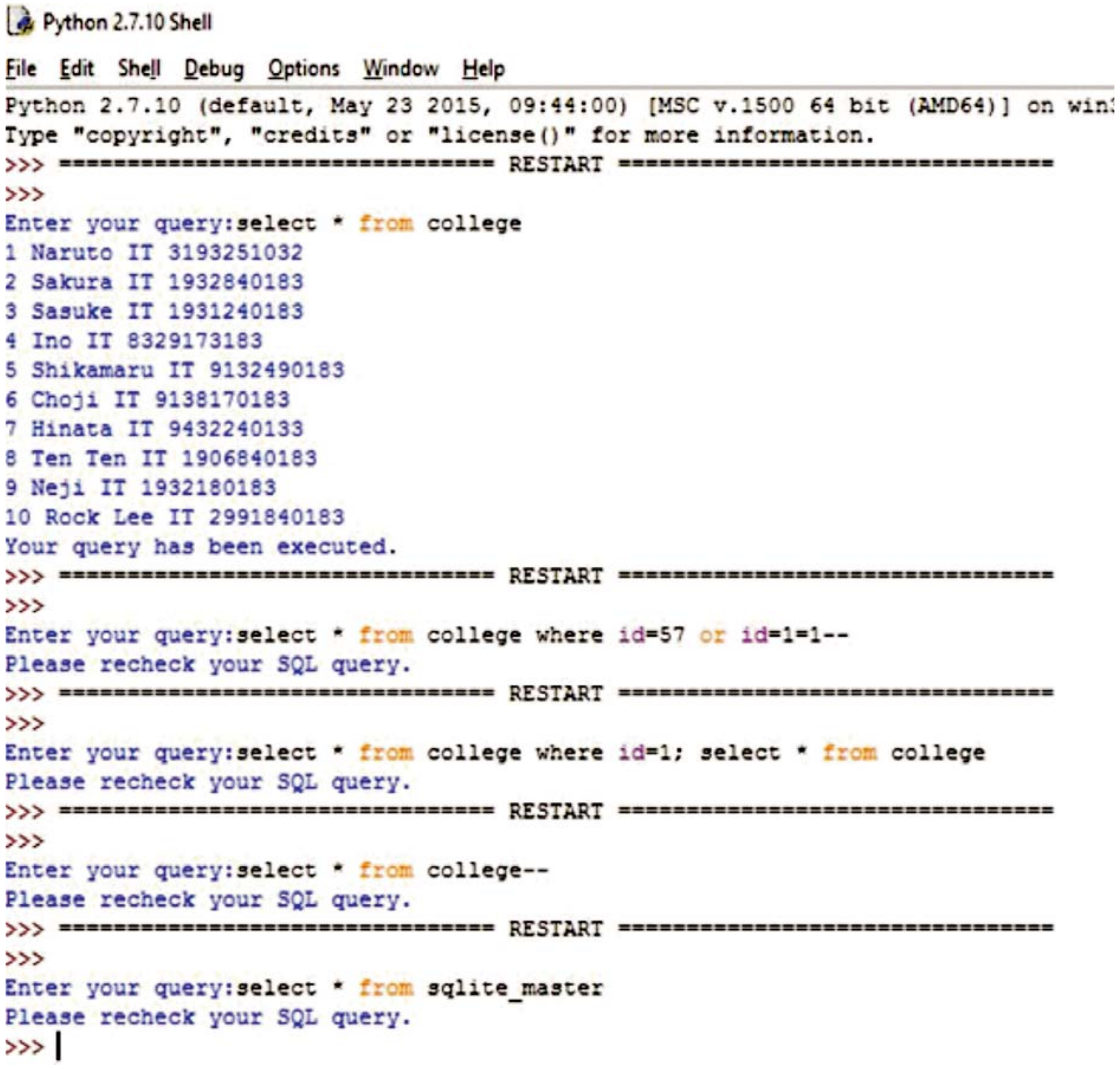
Table 1
Blacklist of Vaccine

Blacklist Rule	*Reason*
1=1--	Is true for all rows so attacker gets all the rows from database
admin'--	Logs in as admin since rest of SQL query is ignored
admin' #	Logs in as admin since rest of SQL query is ignored
admin'/*	Logs in as admin since rest of SQL query is ignored
' or 1=1#	Common log-in trick
' or 1=1/*	Common log-in trick
') or '1'='1--	Common log-in trick
') or ('1'='1--	Common log-in trick
;	Allows stacked queries
--	Comments out rest of the query
||	Allows concatenation of queries
CASE	Allows adding an IF statement
sqlite_master	Lists the schema of database

The first blacklist phrase is a common log-in trick that returns all rows because 1=1 is always true. The next three phrases are used to log-in to the system as the administrator and also comment out the rest of the query to enter the administrator account. The four queries following that are also common log-in attacks similar to the first phrase. The next four phrases are symbols that can allow stacked queries, comment out legitimate part of the query and allow concatenation of erroneous queries. Therefore they are blocked. The last phrase is a command where the schema of the database is listed. The same activity can be carried out using another command, therefore, this keyword is blocked.

4. RESULT

The program on running will prompt the user to enter a query. After it has been entered, the query is checked and then allowed to access the database. A screen-shot of sample input and output through Vaccine can be seen in Fig. 2.



```
Python 2.7.10 (default, May 23 2015, 09:44:00) [MSC v.1500 64 bit (AMD64)] on win
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Enter your query:select * from college
1 Naruto IT 3193251032
2 Sakura IT 1932840183
3 Sasuke IT 1931240183
4 Ino IT 8329173183
5 Shikamaru IT 9132490183
6 Choji IT 9138170183
7 Hinata IT 9432240133
8 Ten Ten IT 1906840183
9 Neji IT 1932180183
10 Rock Lee IT 2991840183
Your query has been executed.
>>> ================================ RESTART ================================
>>>
Enter your query:select * from college where id=57 or id=1=1--
Please recheck your SQL query.
>>> ================================ RESTART ================================
>>>
Enter your query:select * from college where id=1; select * from college
Please recheck your SQL query.
>>> ================================ RESTART ================================
>>>
Enter your query:select * from college--
Please recheck your SQL query.
>>> ================================ RESTART ================================
>>>
Enter your query:select * from sqlite_master
Please recheck your SQL query.
>>> |
```

Figure 2: Output after entering a proper legitimate query and four queries containing blacklisted phrases

Fig 2. shows the output of multiple queries that have been executed through the system. The first query contains no malicious phrases and hence it is allowed to access the database and carry out the required operations. Therefore it displays all the rows of the database. The other queries that follow all contain at least one erroneous keyword, therefore, those queries are not allowed to access the database and an error message is displayed. In the second query, there is no ID numbered 57 so the other half of the statement is checked. Since 1=1 is always true, all the rows would have been returned. But this has been successfully blocked by the vaccine. The third query uses a semicolon for executing multiple statements by the hacker, so that has been blocked. In the fourth query, the attacker tries to use comments to ignore the rest of legitimate code, so that has also been blocked by Vaccine. In the last query, the keyword sqlite_master can be used by the hacker to gather details of the database. Therefore that has also been blocked.

5. CONCLUSION AND SUMMARY

The databases contain vital information which should be protected from malicious users. If unchecked, it can cause danger to the innocent users of the application in many different ways. We have developed a software program that checks such malicious queries which are known as SQL injection attacks. The program does not allow such queries to be passed to the database. Hence this will be a useful addition to mobile applications.

6. ACKNOWLEDGMENTS

Dr. R. Subburaj, thank you for guiding me throughout the course of this project and helping me to publish this paper.

REFERENCES

[1] https://www.owasp.org/index.php/SQL_Injection

[2] Sabo Sambath and Egui Zhu, “Frontiers in Computer Education”, pp. 502, 27-Feb-2012.

[3] Inyong Lee, Soonki Jeong, Sangsoo Yeo, Jongsub Moon, “A novel method for SQL injection attack detection based on removing SQL query attribute values”, Mathematical and Computer Modelling

[4] Indrani Balasundaram, Dr. E. Ramaraj, “An approach to prevent and detect SQL attacks in database using web service”, International Journal of Computer Science and Network Security

[5] Kanchana Natarajan, Sarala Subramani, “ Generation of SQL-injection free secure algorithm to detect and prevent SQL-injection attacks “, Procedia Technology

[6] XuePing-Chen, “SQL injection attack and guard technical research”, Procedia Engineering

[7] https://www.owasp.org/index.php/Mobile_Top_10_2014-M7

[8] William G.J. Halfond, Jeremy Viegas and Alessandro Orso, “A Classification of SQL Injection Attacks and Countermeasures”, Proceedings of the IEEE

[9] Atefeh Tajpour, Suhaimi Ibrahim, Maslin Masrom, “SQL injection detection and prevention techniques ”, International Journal of Advancements in Computing Technology

[10] Subburaj Ramasamy, Anuj Singh, Deepak Singal, “Enhancing the Security of C/C++ Programs using Static Analysis,” Indian Journal of Science and Technology

[11] https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/#top

[12] http://atta.cked.me/home/sqlite3injectioncheatsheet