SECURE QUERY PROCESSING BY BLOCKING SQL INJECTION ATTACK (SQLIA)

Mohammed Firdos Alam Sheikh
Pacific Institute of Technology,Udaipur
firdos.sheikh@gmail.com

ABSTRACT

SQL Injection Attacks (SQLIAs) are one of the topmost threats for web application security, and SQL injections are one of the most serious vulnerability types. Recently the incident of SQLIA is so high. They allow attackers to obtain unauthorized access to the databases. With SQL injections, cyber-criminals can take complete remote control of the database, with the consequence that they can become able to manipulate the database to do anything they wish. SQL Injection attack techniques have become more common, more ambitious, and increasingly sophisticated, so there is a deep need to find an effective and feasible solution for this problem in the computer security community. Although many researchers and practitioners have proposed various methods to address the SQL injection problem, current approaches either fail to address the full scope of the problem or have limitations that prevent their use and adoption. a Detection or prevention of SQLIAs is a topic of active research in the industry and academia. To achieve those purposes, automatic tools and security systems have been implemented, but none of them are complete or accurate enough to guarantee an absolute level of security on web applications. One of the important reasons of this shortcoming is that there is a lack of common and complete methodology for the evaluation either in terms of performance or needed source code modification which in terms is an over head for an existing system. So we feel that there should be such type of mechanism which will be easily deployable, does not need source code modification as well as provide a good performance. To address this problem, we present review of the different types of SQL injection attacks, For each type of attack, we provide descriptions and examples of how attacks of that type could be performed. We also present and analyze existing detection and prevention techniques against SQL injection attacks.

KEYWORDS

SQL Injection, Security, Bypassing, Queris, Stored Procedure

1. INTRODUCTION

SQL injection vulnerabilities have been described as one of the most serious threats for Web applications [3,11].With SQL injections, cyber-criminals can take complete remote control of the database, with the consequence that they can become able to manipulate the database the resulting security violations can include identity theft, loss of confidential information, and fraud. Web applications that are vulnerable to SQL Injection Attacks (SQLIAs) are widespread-a study by Gartner Group on over 300 Internet Web sites has shown that most of them could be vulnerable to SQLIAs.
To address this problem, developers have proposed a range of coding guidelines (e.g., [18]) that promote defensive coding practices, such as encoding user input and validation.
many proposed solutions fail to address the full scope of the Problem.Therefore, most of the solutions proposed detect or prevent only a subset of the possible SQLIAs. To address this problem, we present a survey of SQL injection attacks known to date. To compile the survey, we used information gathered from various sources, such as papers, Web sites, mailing lists, and experts in the area. For each attack type considered, we give a characterization of the attack, illustrate its effect, and provide examples of how that type of attack could be performed. This set of attack types is then used to evaluate state of the art detection and prevention techniques and compare their strengths and weaknesses.The results of this comparison show the effectiveness of these techniques.
The rest of this paper is organized as follows: Section 2 provides How SQLIAs work, SQL Injection Mechanism and related concepts. Section 3 defines and presents the different attack types. Sections 4 provide review and techniques to prevent SQLIAs. Finally, we provide summary and conclusions in Sect- 5.

2. HOW SQL INJECTION ATTACKS (SQLIA) WORK

An *SQL Injection Attack (SQLIA)* occurs when an attacker changes the SQL query by inserting new SQL keywords or operators into the query. In this section, we define two important characteristics of SQLIAs that we use for describing attacks: injection mechanism and attack intent.

SQL injection refers to a class of code-injection attacks in which data provided by the user is included in an SQL query in such a way that part of the user's input is treated as SQL code. By leveraging these vulnerabilities, an attacker can submit SQL commands directly to the database.

2.1 Consequences of SQL Injections Attacks

With SQL injections, cyber-criminals can take complete remote control of the database, with the consequence that they can become able to manipulate the database to do anything they wish, including:

- Insert a command to get access to all account details in a system, including user names and passwords.
- Shut down a database
- Upload files
- Through reverse lookup, gather IP addresses and attack those computers with an injection attack.
- Corrupting, deleting or changing files and interact with the OS, reading and writing files
- Online shoplifting e.g. changing the price of a product or service, so that the cost is negligible or free
- Delete the database and all its contents

2.2 The SQL Injection Mechanisms

Malicious SQL statements can be introduced into a vulnerable application using many different input mechanisms. In this section, we explain the most common mechanisms.

Injection through cookies: Cookies are those files that contain state information that are generated by Web applications and stored on the client machine. When a client returns to a Web application, cookies can be used to restore the client's state information. Since the client has control over the storage of the cookie, a malicious client could tamper with the cookie's contents. If a Web application uses the cookie's contents to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie [5].

Injection through user input: The attackers inject SQL commands by providing suitably crafted user input. In most SQLIAs that target Web applications, user input typically comes from form submissions that are sent to the Web application via HTTP GET or POST requests [14].Web applications are generally able to access the user input contained in these requests as they would access any other variable in the environment.

Injection through server variables: Server variables are a collection of variables that contain HTTP, network headers, and environmental variables. Web applications use these server variables in a variety of ways, such as logging usage statistics and identifying browsing trends. If these variables are logged to a database without sanitization, this could create an SQL injection vulnerability. Because attackers can forge the values that are placed in HTTP and network headers, they can exploit this vulnerability by placing an SQLIA directly into the headers.When the query to log the server variable is issued to the database, the attack in the forged header is then triggered.

Second-order injection: In second-order injections, attackers seed malicious inputs into a system or database to indirectly trigger an SQLIA when that input is used at a later time. The objective of this kind of attack differs significantly from a regular (i.e., first- order) injection attack. Second-order injections are not trying to cause the attack to occur when the malicious input initially reaches the database. Instead, attackers rely on knowledge of where the input will be subsequently used and craft their attack so that it occurs during that usage. To clarify, we present a classic example of a second order injection attack (taken from [1]). In the exam- ple, a user registers on a website using a seeded user name, such as “admin’ -- ”. The application properly escapes the single quote in the input before storing it in the database, preventing its potentially malicious effect. At this point, the user modifies his or her password, an operation that typically involves (1) checking that the user knows the current password and (2) changing the pass- word if the check is successful. To do this, the Web application might construct an SQL command as follows:

queryString="UPDATE users SET password=’" + newPassword + "’ WHERE userName=’" + userName + "’ AND password=’" + oldPassword + "’"

newPasswordand oldPasswordare the new and old pass- words, respectively, and userNameis the name of the user cur- rently logged-in (i.e., ‘‘admin’--’’). Therefore, the query string that is sent to the database is (assume that newPasswordand oldPas-swordare “newpwd” and“oldpwd”):

UPDATE users SET password=’newpwd’

WHERE userName= ’admin’--’ AND password=’oldpwd’

Because “--” is the SQL comment operator, everything after it is

ignored by the database. Therefore, the result of this query is that the database changes the password of the administrator (“admin”) to an attacker-specified value.

Second-order injections can be especially difficult to detect and prevent because the point of injection is different from the point where the attack actually manifests itself. A developer may prop- erly escape, type-check, and filter input that comes from the user and assume it is safe. Later on, when that data is used in a dif- ferent context, or to build a different type of query, the previously sanitized input may result in an injection attack.

2.3 Attack Intention

When a threat agent utilizes a crafted malicious SQL input to launch an attack, the attack intention is the goal that the threat agent tries to achieve once the attack has been successfully executed.

Identifying Inject-able Parameters [2]: Inject-able parameters are the parameters or the user input fields of the Web applications directly used by server-side program logic to construct SQL statements, which are vulnerable to SQLIA. In order to launch a successful attack, a threat agent must first discover which parameters are vulnerable to SQL injection attack.

Performing database finger-printing [2]: Database finger-print is the information that identifies a specific type and version of database system. Every database system employs a different proprietary SQL language dialect. For example, the SQL language employed by Microsoft SQL server is T-SQL while Oracle SQL server uses PL/SQL. In order for an attack to be succeeded, the attacker must first find out the type of and version of database deployed by a web application, and then craft malicious SQL input accordingly.

Bypassing Authentication [2]: Authentication is a mechanism employed by web application to assert whether a user is who he/she claimed to be. Matching a user name and a password stored in the database is the most common authentication mechanism for web applications. Bypassing authentication enables an attacker to impersonate another application user to gain un-authorized access.

Determining database schema [2]: Database schema is the structure of the database system. The schema defines the tables, the fields in each table, and the relationships between fields and tables. Database schema is used by threat agents to compose a correct subsequent attack in order to extract or modify data from database.

Adding or Modifying Data [2]: Database modification provides a variety of gains for a threat agent, for instance, a hacker can pay much less for a online purchase by altering the price of a product in the database.

Extracting Data [2]: In most of the cases, data used by web applications are highly sensitive and desirable to threat agents. Attacks with intention of extracting data are the most common type of SQL injection attacks.

Evading detection [2]: This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms.

Performing denial of service [2]*:* These attacks are performed to shut down the database of a Web application, thus denying service to other users. Attacks involving locking or dropping database tables also fall under this category.

Executing Remote Commands [2]: Remote commands are executable code resident on the compromised database server. Remote command execution allows an attacker to run arbitrary programs on the server. Attacks with this type of intention could cause entire internal networks being compromised.

3. METHODOLOGY FOR A SUCCESSFUL SQLIA

In this section, we present and discuss the different kinds of SQLIAs known to date. For each attack type, we provide a descriptive *name*, one or more *attack intents*, a *description* of the attack, an attack *example*, and a set of *references* to publications and Web sites that discuss the attack technique and its variations in greater detail.

Tautologies

Attack Intent: Bypassing authentication, identifying injectable pa- rameters, extracting data.

Description: The general goal of a tautology-based attack is to in- ject code in one or more conditional statements so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. The most com- mon usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an injectable field that is used in a query’s WHEREconditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the query to be returned. In general, for a tautology-based attack to work, an attacker must consider not only the injectable/vulner-able parameters, but also the coding constructs that evaluate the query results. Typically, the attack is successful when the code ei- ther displays all of the returned records or performs some action if at least one record is

returned.
Example: In this example attack, an attacker submits “or 1=1- -
for the *login* input field (the input submitted for the other fields is irrelevant). The resulting query is:

SELECT status FROM users WHERE user name=’’ or 1=1 -- AND
pass=’’

The code injected in the conditional (OR 1=1) transforms the entire WHERE clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them.
References:[1,13, 15,12]

End of Line Comment:

Attack Intent: Bypassing authentication

After injecting code into a particular field, legitimate code that follows are nullified through usage of end of line comments. An example would be to add [- -] after inputs so that remaining queries are not treated as executable code, but comments. This is useful since threat agents may not always know the syntax or fields in the server.
Example: In this example attack, an attacker submits [admin'--] for the *user name* input field (the input submitted for the other fields is irrelevant). The resulting query is:

SELECT * FROM user WHERE username = 'admin'--' AND password = ‘’

The code injected in the condition [admin‟ -] transform the WHERE clause in that way it is going to log the attacker as admin user if there is a user name called “admin”, because rest of the SQL query will be ignored.

Union Query

Attack Intent: Bypassing Authentication, extracting data. *Description*: In union-query attacks, an attacker exploits a vulner- able parameter to change the data set returned for a given query. With this technique, an attacker can trick the application into re- turning data from a table different from the one that was intended by the developer. Attackers do this by injecting a statement of the form: `UNION SELECT <rest of injected query>`. Because the attackers completely control the second/injected query, they can use that query to retrieve information from a specified table.The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

Example: [" UNION SELECT username, password from user_info where user_name=" abc" - -] into the login field, which produces the following query:

SELECT * FROM users WHERE login='' UNION SELECT password from user_info where user_name='abc'-- AND pass=''

Assuming that there is no login equal to ”, the original first query returns the null set, whereas the second query returns data from the “user_info” table. In this case, the database would return column “password” for username “abc.” The database takes the results of these two queries, unions them, and returns them to the application. In many applications, the effect of this operation is that the value for “password” is displayed along with the user information.

References: [1, 13, 15]

Piggy-Backed Queries

Attack Intent: Extracting data, adding or modifying data, perform- ing denial of service, executing remote commands.

Description: In this attack type, an attacker tries to inject additional queries into the original query. We distinguish this type from others because, in this case, attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that “piggy-back” on the original query. As a result, the database receives multiple SQL queries. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first.This type of attack can be extremely harmful. If successful, attackers can insert virtually any type of SQL command, including stored procedures, into the additional queries and have them executed along with the original query. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string.

Example: If the attacker inputs “’; drop table users - -” into the *pass* field, the application generates the query:

```
SELECT accounts FROM users WHERE login='doe' AND
pass=''; drop table users -- ' AND pin=123
```

After completing the first query, the database would recognize the

Stored procedures are routines stored in the database and run by the database engine. These procedures can be either user-defined procedures or procedures provided by the database by default query delimiter (“`;`”) and execute the injected second query. The result of executing the second query would be to drop table `users`, which would likely destroy valuable information. Other types of queries could insert new users into the database or execute stored procedures. Note that many databases do not require a special character to separate distinct queries, so simply scanning for a query separator is not an effective way to prevent this type of attack.

References: [1, 15, 12]

Stored Procedures

Attack Intent: Performing privilege escalation, performing denial of service, executing remote commands.

Description: SQLIAs of this type try to execute stored procedures present in the database. Today, most database vendors ship databases with a standard set of stored procedures that extend the function- laity of the database and allow for interaction with the operating system. Therefore, once an attacker determines which backend- database is in use, SQLIAs can be crafted to execute stored proce- dures provided by that specific database, including procedures that interact with the operating system.

It is a common misconception that using stored procedures to write Web applications renders them invulnerable to SQLIAs. De- velopers are often surprised to find that their stored procedures can be just as vulnerable to attacks as their normal applications [12, 14]. Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run arbitrary code on the server or escalate their privileges [6].

```
CREATE PROCEDURE DBO.isAuthenticated
    @userName varchar2, @pass varchar2, @pin int
AS
    EXEC("SELECT accounts FROM users
    WHERE login=’" +@userName+ "’ and pass=’" +@password+ "’ and pin=" +@pin);
GO
```

Figure 1: Stored procedure for checking user Authorization.

Example: This example demonstrates how a parameterized stored procedure can be exploited via an SQLIA. In the example, we as- sume that the query string constructed at lines 5, 6 and 7 of our

example has been replaced by a call to the stored procedure de- fined in Figure 2. The stored procedure returns a true/false value to indicate whether the user's credentials authenticated correctly. To launch an SQLIA, the attacker simply injects " ' ; SHUTDOWN; -
-" into either the userNameor passwordfields. This injection causes the stored procedure to generate the following query:

SELECT accounts FROM users WHERE
login='doe' AND pass=' '; SHUTDOWN;

At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second, malicious query is executed, which results in a database shut down. This example shows that stored procedures can be vulnerable to the same range of attacks as traditional application code.
References: [1, 4, 6, 7, 14, 15, 13, 12]

4. PREVENTION OF SQLIAS

Researchers have proposed a wide range of techniques to address the problem of SQL injection. These techniques range from devel- opment best practices to fully automated frameworks for detecting and preventing SQLIAs. In this section, we review these proposed techniques and summarize the advantages and disadvantages asso- ciated with each technique.

Framework Support: Recent frameworks for a web-applications provide functionality that can be used to prevent SQL injections. An input validator prohibits user input from including meta-characters to avoid SQL injections. But if we want to include meta- characters in the input we can not prevent SQL injections.provide a functionality that can be used to prevent SQL injections.
Prepare Statement [22]: SQL provides the prepare statement, which separates the values in a query from the structure of SQL. The programmer defines a skeleton of an SQL query and then fills in the holes of the skeleton at runtime. The prepare statement makes it harder to inject SQL queries because the SQL structure can not be changed. To use the prepare statement, we must modify the web application entirely all the legacy web applications must be re-written to reduce the possibility of SQL injections.

Static Analysis [16]: Wassermann and Su proposed an approach that uses a static analysis combined with automated reasoning. This technique verifies that the SQL queries generated in the application usually do not contain a tautology. This technique is effective only for SQL injections that insert a tautology in the SQL queries, but can not detect other types of SQL injections attacks.

Machine Learning Approach [17]: Valeur *et al.* proposed the use of an intrusion detection system (IDS) based on a machine learning technique. IDS is trained using a set of typical application queries, builds models of the typical queries, and then monitors the application at runtime to identify the queries that do not match the model. The overall IDS quality depends on the quality of the training set a poor training set would result in a large number of false positives and negatives.

Instruction-Set Randomization [18]: SQLrand provides a framework that allows developers to create SQL queries using randomized keywords instead of the normal SQL keywords. A proxy between the web application and the database intercepts SQL queries and de-randomizes the keywords. The SQL keywords injected by an attacker would not have been constructed by the randomized keywords, and thus the injected commands would result in a syntactically incorrect query. Since SQLrand uses a secret key to modify keywords, its security relies on attackers not being able to discover this key. SQLrand requires the application developer to rewrite code.

Taint-Based Technique [19]: Pietraszek and Berghe modified a PHP interpreter to track taint information at the character level. This technique uses a context-sensitive analysis to reject SQL queries if an un-trusted input has been used to create certain types of SQL tokens. A common drawback of this approach is that they require modifications to the runtime environment, which diminishes the portability.

Combined Static and Dynamic Analysis: Su *et al.* [20] present grammar-based approach to detect and stop queries having SQLIAs by implementing SQLCHECK tool. They mark user supplied portions in queries with a special symbol and augment the standard SQL grammar with production rule. A parser is generated based on the augmented grammar. The parser successfully parses the generated query at runtime, if there are no SQLIAs in the generated queries after adding user inputs. This approach uses a secret key to discover user inputs in the SQL queries. Thus, the security of the approach relies on attackers not being able to discover the key. Additionally, this approach requires the application developer to rewrite code to manually insert the secret keys into dynamically generated SQL queries. Buehrer et al [21]. Secure vulnerable SQL statements by comparing the parse tree of a SQL statement before and after input and only allowing SQL statements to execute if the parse trees match. They conducted a study using one real world web application and applied their SQLGUARD solution to each application. They found that their solution stopped all of the SQLIAs in their test set without generating any false positives. While it stopped all of the SQLIAs, their solution required the developer to rewrite all of their SQL code to use their custom libraries.

5. CONCLUSION

SQL injection is a common technique hackers employ to attack these web-based applications. These attacks reshape the SQL queries, thus altering the behavior of the program for the benefit of the hacker. In this paper, we present a technique for detecting and preventing SQLIA incidents.In this paper, we have presented a survey and comparison of current techniques for detecting and preventing SQLIAs. To perform this evaluation, we first identified the various types of SQLIAs known to date. We then evaluated the considered techniques in terms of their ability to detect and/or prevent such attacks. We also studied the different mechanisms through which SQLIAs can be introduced into an application and identified which techniques were able to handle which mechanisms. Lastly, we summarized evaluated to what extent its detection and prevention mechanisms could be fully automated.

Our evaluation found several general trends in the results. Many of the techniques have problems handling attacks that take advantage of poorly-coded stored procedures and cannot handle attacks that disguise themselves using alternate encodings.

Future evaluation work should focus on evaluating the techniques' precision and effectiveness in practice. Empirical evaluations such as those presented in related work (e.g., [11]) would allow for comparing the performance of the different techniques when they are subjected to real-world attacks and legitimate inputs.

6. ACKNOWLEDGEMENTS

This work was partially supported by Many people directly or indirectly. Mrs.Deepti Nathawat provided valuable feedback and suggestions that helped improve the paper.

7. REFERENCES

[1] C. Anley. Advanced SQL Injection In SQL Server Applications.White paper, Next Generation Security Software Ltd., 2002.

[2] W. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures.*Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[3] D. Aucsmith. Creating and Maintaining Software that Resists Malicious Attack. http://www.gtisc.gatech.edu/bio aucsmith.html, September 2004. Distinguished Lecture Series.

[4] F. Bouma. Stored Procedures are Bad, O'kay? Technical report, Asp.Net Weblogs, November 2003. http://weblogs.asp.net/fbouma/archive/2003/11/18/38178.aspx.

[5] M. Dornseif. Common Failures in Internet Applications,May 2005.http://md.hudora.de/presentations/2005-common-failures/dornseif-common-failures-2005-05-25.pdf.

[6] E. M. Fayo. Advanced SQL Injection in Oracle Databases. Technical report, Argeniss Information Security, Black Hat Briefings, BlackHat USA, 2005.

[7] P. Finnigan. SQL Injection and Oracle - Parts 1 & 2. Technical Report, Security Focus, November 2002. http://securityfocus.com/infocus/1644

[8] T. O. Foundation. Top Ten Most Critical Web Application Vulnerabilities, 2005. http://www.owasp.org/documentation/topten.html.

[9] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 04) – Formal Demos*, pages 697–698, 2004.

[10] N. W. Group. RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1.Request for comments, The Internet Society, 1999.

[11] W. G. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA2005), pages 22–28, St. Louis, MO, USA, May 2005.

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

[13] S. Labs. SQL Injection. White paper, SPI Dynamics, Inc., 2002.

http://www.spidynamics.com/assets/documents/ WhitepaperSQLInjection.pdf.

[14] 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.

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

[16] G.Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. *In Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 70–78, 2004.

[17] 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 andVulnerability Assessment (DIMVA)*, pages 123–140, 2005.

[18] S.Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. *In Proceedings of the Applied Cryptography and Network Security (ACNS)*, pages 292–304, 2004.

[19] T. Pietraszek and C. Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. *In Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 124–145, 2005.

[20] Z.Su and G.Wassermann. The Essence of Command Injection Attacks in Web Applications. Annual Symposium onPrinciple of Programming Languages (POPL), pages372–382, 2006.

[21] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," 5th International Workshop on Software Engineering and Middleware, pages 106-113, 2005

[22] Stephen Thomas, Laurie Williams. *Using Automated Fix Generation to Secure SQL Statements.* Third International Workshop on Software Engineering for Secure Systems (SESS'07), pages 9-9,May 2007.