ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 10, October 2015

Web Intrusion Using Advanced SQL Injector and Countermeasures

Mrs. R. Raghavi Tharani[1], S. Jayasurya[2], A. Azath[3]

Assistant Professor, Dept. of BCA&SS, Sri Krishna Arts and Science College, Coimbatore, India[1]

P.G. Student, Dept. of BCA&SS, Sri Krishna Arts and Science College, Coimbatore, India[2, 3]

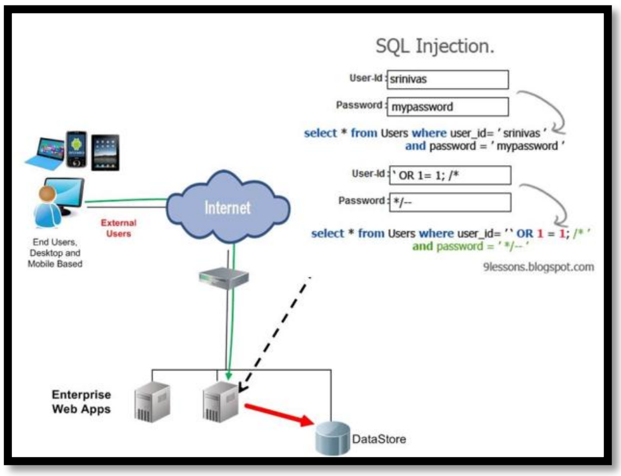
Abstract: The topic web intrusion using advanced SQL injector and counter measures, SQL injection has become a predominant type of attack that target web applications. It allows attackers to obtain unauthorized access to the back-end database to alter the intended application-generated SQL queries. Researchers have proposed various solutions to address SQL injection problems. Our dependence on the web applications for the fulfillment of our daily needs (like share trading, banking, ticket booking, online shopping, payment of bills etc.) has increased. Because of this, our private data is present in the databases of various applications on Web. The defense of this myriad amount of data is a theme of major anxiety. In current times, SQL Injection attacks have emerged as a major risk to database security. In this paper we characterize SQL Injections, illustrate how SQL Injections will perform. In addition we have also surveyed the various SQL Injection recognition and anticipation tools and well-known assail methods.

Keywords: Introduction, Injection Mechanism, tools.

1. INTRODUCTION:

SQL injection is one of the main technique attackers use to negotiate a database. This type of attack use vulnerabilities accessible in web applications or stored procedures in the back-end database server. It allow attackers to inject crafted cruel SQL query segment to change the planned effect of a SQL query, so that attackers can obtain illegal access to a database, read or modify data, make the data occupied to other users, or even damage the database server. According to a review report released in 2010 by the IBM X-Force® RD team, the number of SQL injection attacks has enlarged rapidly in recent years, and SQL injection has become the major type of attacks that target web applications. During the initial half of the year of 2010, the average sum of daily SQL injection attacks around the world is about 400,000.Web applications and their essential databases require not only cautious configuration and programming to guarantee security, but also effective protection mechanisms to avoid attacks. Researchers have planned various solution and techniques to address the SQL injection problems. On the other hand, there is no one solution that can guarantee complete security. Many current solutions often cannot address all of the harms. For example, many techniques anticipated are based on the postulation that only the SQL statements that receive user input are at risk to SQL injection attacks.

SQL (Structured Query Language) is a general language worn to insert, update, retrieve and delete information from the databases. When we penetrate our information (like login identification etc.) in the input field provided on the web form of a Web Application, it form the part of the SQL query wrote at the backend, to be perform on the database. For instance, when we login into our mailbox, we present user id and password. The user id and password makes the part of the interior SQL query. Then the SQL query is executed on the database to test whether the login credentials presented match with those there in the tables on the database. The attacker, who is not aware of the login identification but, wants to gain admission to the mailbox by unmerited means, provides SQL code in its place of correct input in the test fields of the web form. This cruel code changes the structure of the original SQL query and as a result, allows the attacker to achieve access to the information it was not certified for.



```
select * from Users where user_id= ' srinivas '
                   and password = ' mypassword '

select * from Users where user_id= '' OR 1 = 1; /* '
                   and password = ' */-- '
```

Fig1. Sql Injection Architecture

2. INJECTION MECHANISMS

Cruel SQL statements can be introduced into a defenseless application using many diverse input mechanisms. In this section, we clarify the most common mechanisms. Injection through the user input: In this type of injection, attackers inject SQL commands by providing properly crafted user input. A Web application can interpret user input in numerous ways on the basis of environment in which the application is organized. In most SQLIAs that intention Web applications, user input classically comes from form compliance that are propelled to the Web application via POST requests or HTTP GET. Web applications are generally able to contact the user input contained in these needs as they would contact any other variable in the atmosphere. Injection through cookies: Cookies are files that contain state information produced by Web applications and piled up on the client machine. When a client returned to a Web application, cookies are used to renovate the client's state information. Since the client has power over the storage of the cookie, a cruel client could tamper with the cookie's contents. If Web applications make use of the cookie's contents to construct SQL queries, an attacker could easily submit an attack by embedding it in the cookie.

Injection through server variable: Server variables are a collection of variables that hold environmental variables, network headers and HTTP. Web applications use these server variables in many ways, such as logging practice statistics and identifying browsing trend. If these variables are logged to a database without purification, this could generate SQL injection susceptibility. Because attackers can counterfeit the values that are positioned in HTTP and network headers, they can exploit this susceptibility by placing an SQLIA straight into the headers. When the query to log the server variable is mattered to the database, the attack in the forged header is then activated.

Second-order injection: In this injections, attackers start cruel inputs into a system or database to ultimately activate an SQLIA when that input is worn at a later time. The purpose of this kind of attack differs significantly from a regular injection attack. Second-order injections are not annoying to cause harass to occur when the cruel input initially reaches the database. Instead, attackers depend on knowledge of where the input will be subsequently used and skill their attack so that it occur at some stage in that practice. To clarify, we present a classic example of a next order injection attack (taken from). In the example, a user registers on a website using a sowed user name, such as "admin' --". The application correctly escapes the solo quote in the input before accumulate it in the database, preventing its potentially cruel effect. At this point, the user alters his or her password, a process that naturally involves (1) examining that the user knows the recent password and altering the password if the check is triumphant. To do this, the Web application might build an SQL command as follow:

**queryString="UPDATE users SET password='" + newPassword +
"' WHERE userName='" + userName + "' AND password='" +
oldPassword + "'"newPassword and oldPassword are the new and old passwords, respectively, and userName is the name of the user currently logged-in (i.e., ''admin'--'').[1]**

Therefore, the query string that is sent to the database is

(assume that newPassword and oldPas-sword are "newpwd" and"oldpwd"):

UPDATE users SET password='newpwd' WHERE userName= 'admin'--' AND password='oldpwd'[2]

Because "--" is the SQL commentary operator, everything after it is unnoticed by the database. Therefore, the effect of this query is that the database will change the password of the admin ("administrator") to an attacker-specified value. Second-order injections can be especially hard to detect and prevent because the position of injection is different from the position where the attack really apparent itself. A developer may correctly escape, type-check, and strain input that comes from the client and assume it is secure. Later on, when that data is used in a dissimilar context, or to build a diverse type of query.

CANDID	It is a instrument developed to safeguard Web applications in Java language next to SQL Injection attacks. It uses candidate inputs to dynamically assume about the programmer intended query structure. Candid consists of two apparatus: an online SQL parse tree checker and an offline Java program transformer.
AMNESIA	Detection and anticipation technique, which uses fixed and dynamic study in combination. During static analysis, it predicts the legitimate queries that can be generated by the application. During dynamic analysis, it uses runtime examine to check the queries generated in static analysis against the actual set of produced query.
Positive Tainting	Dynamic method to detect and prevent SQL injections by performing dynamic spoil Firstly, it finds and highlight the belief data. Then it executes accurate spoil propagation by highlighting the belief data at character level. Finally, it execute syntax-aware
SQL Rand	The concept of Instruction-Set randomization is practical to the SQL language to notice and abort query which hold injected code. Here, each SQL keyword is joined with a random digit to mislead the invader.
SQL DOM	Object oriented model in which SQL queries are generated by influencing objects which are strongly-typed to the database. It inspects the dynamically produced query at of compile time.

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 10, October 2015

Viper	An instrument used for Web Application penetration testing which uses heuristic advance for detecting SQL Injections.
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.
ADMIRE	It is a danger risk replica which give a thorough and step-by-step method to identify and sensible the effect of SQL Injections.
WAVES	A Black box technique which searches for vulnerable locations in a Web application using a Web flatterer and then constructs attacks which target these locations. Finally, it watches the response of the Web application to these assails using machine learning technique.
JDBC-Checker	It is a static checking technique which tests for the rightness of the dynamically-generated SQL query

Table 1: SQL Tools

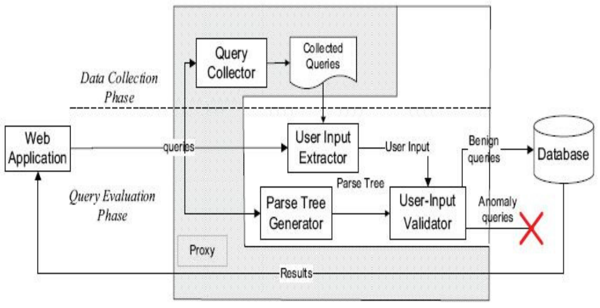


Fig 1: Architecture of SQL Injection and Problems

3. PREVENTION OF SQL I

Researchers have planned a wide range of technique to address the difficulty of SQL injection. These techniques sort from development best practices to completely automated frameworks for spotting and averting SQLIAs. In this section, we review these proposed techniques and summarize the merits and demerits linked with each method.

4. DETECTION AND PREVENTION TECHNIQUES

Researchers have planned a range of method to assist developers and pay off for the inadequacy in the application of suspicious coding.
Black Box Testing. Huang and classmates propose WAVES, a black-box method for testing Web applications for SQL injection vulnerabilities. The method uses a Web crawler to recognize all points in a Web application that can be worn to inject SQLIAs. It then constructs attacks that aim such points based on a specified list of pattern and attack techniques. WAVES then check the application's response to the attacks and uses machine learning techniques to improve its assail method. This technique improves over most penetration testing techniques by using machine learning approach to direct its testing. However, similar to all black-box and penetration testing techniques, it cannot provide guarantees of completeness.
Static Code Checker. JDBC-Checker is a method for statically checking the type rightness of dynamically-generated SQL queries. This method was not developed with the intent of detecting and preventing general SQLIAs, but can yet be used to avert attacks that take advantage of type mismatches in a dynamically-produced query string. JDBC-Checker is capable to detect one of the root cause of SQLIA vulnerabilities in code— rude type checking of input. On the other hand, this method would not catch more common forms of SQLIAs because most of this attack consist of syntactically and type correct query.

Wassermann and Su suggest a method that uses static analysis joined with automated reasoning to validate that the SQL query produced in the application layer cannot hold a tautology. The main drawback of this method is that its scope is restricted to detecting and preventing tautologies and cannot notice other types of attack.

Combined Static and Dynamic study. AMNESIA is a model-based method that combines both runtime monitoring and static analysis. In its static segment, AMNESIA uses static analysis to build model of the different types of query an application can lawfully produce at each point of contact to the database. In its dynamic segment, AMNESIA intercepts all query before they are sent to the database and check each query against the statically-built model. Query that infringes the model are identified as SQLIAs and banned from executing on the database. In their evaluation, the authors have exposed that this method perform well against SQLIAs. The primary limitation of this technique is that its success is dependent on the correctness of its static analysis for constructing query models. Certain types of code obfuscation or query development technique could create this step less exact and result in both false positives and false negatives.

Similarly, two recent related approaches, SQL Guard and SQL-Check also check query at runtime to see if they conform to a model of predictable queries. In these approaches, the model is expressed as a grammar that only admits legal queries. In SQL Guard, the model is deduced at runtime by examining the structure of the query before and after the addition of user-input. In SQL Check, the model is specified independently by the developer. Both approaches use a secret key to delimit user input at some point in parsing by the runtime checker, so safety of the approach is dependent on attackers not being able to find out the key. Additionally, the use of these two approaches requires the developer to either rewrite code to use unique intermediate records or physically insert special markers into the code where user input is added to a dynamically generated query.

Several dynamic spoil analysis approaches have been projected. Two alike approaches by Pietraszek and Berghe and Nguyen-Tuong and colleagues modify a PHP interpreter to follow exact per-character spoil information. The technique uses a context sensitive analysis to notice and refuse queries if untrusted input has been used to create certain types of SQL tokens. A ordinary draw-back of these two loom is that they need modification to the runtime environment, which affects portability. A method by Haldar and colleagues and SecuriFly execute a alike approach for Java. However, these techniques do not use the context sensitive analysis engaged by the other two approaches and track spoil information on a per-string foundation. SecuriFly also attempt to sterilize query strings that have been generating using tainted input. However, this purification approach does not help if injection is performing into numeric fields. In common, dynamic taint-based technique have shown a lot

5. TECHNIQUES EVALUATION

In this, we estimate the techniques offered in Section 5 using several different criteria. We first consider which hit types each method is able to address. For the subset of techniques that are based on code improvement, we look at which suspicious coding practice the technique helps enforce. We then identify which injection mechanism each technique is able to handle. Finally, we evaluate the deployment requirements of each technique.

6. EVALUATION WITH RESPECT TO ATTACK TYPES

Intrusion Detection System. Valeur and his colleagues advise the use of an Intrusion Detection System (IDS) to sense SQLIAs. Their IDS format is based on a machine learning method that is trained by means of a set of typical application query. The technique constructs models of the typical query and then monitors the application at runtime to identify query that do not match the model. In their evaluation, Valeur and his colleagues have shown that their system is able to detect attack with a high rate of achievement. Though, the fundamental limitation of learning based techniques is that they can give no guarantees about their discovery abilities because their success is reliant on the quality of the training set used. A reduced training set would cause the learning method to produce a large number of fake positives and negatives.

Proxy Filters. Security Gateway is a proxy filter system that enforces input legalization rules on the data flowing to a Web application. Using their safety Policy Descriptor Language (SPDL), developer give constraint and recognize transformations to be practical to application parameter as they flow from the Web page to the application server. Because SPDL is highly significant, it allows developer substantial freedom in expressing their policy. However, this loom is human-based and, like suspicious programming, requires developers to know not only which data wants to be filtered, but also what pattern and filter to pertain to the data.

Instruction Set Randomization. SQLrand is an loom based on instruction-set randomization. SQLrand provide a framework that allows developers to make queries using randomized instructions in its place of normal SQL keywords. A proxy filter intercept query to the database and de-randomizes the keyword. SQL code injected by an attacker would not build using the randomized instruction set. So, injected commands would effect in a syntactically wrong query. While this technique can be very effectual, it has several practical drawbacks. First, since it uses a secret key to modify orders, security of the approach is reliant on attackers not being able to discover the key. Second, the approaches impose important infrastructure overhead because it require the integration of a proxy for the database in the system.

7. CONCLUSION

SQL injection attack is a serious danger to the growing fame of these applications. The major target of this attack is the database of the Web application and attacker have plan various technique for the same. We have reviewed all the common attack methods and have offered simple illustration for each of them. Also, we have formulate a new solution to counter the difficulty of SQL Injection Attacks but, it is not fool evidence against every well-known attack method. In upcoming we would like to improve our solution so that it can counter all types of attacks.

REFERENCES

[1] IBM Internet Security Systems X-Force® research and development team, "IBM Internet Security Systems™ X-Force® 2009 Mid-Year Trend and Risk Report," Aug. 2009. [Online]. Available: www-935.ibm.com/services/us/iss/xforce/trendreports/. [Accessed: Apr. 10, 2010].

[2] V. Chapela, "Advanced SQL Injection," OWASP Foundation, Apr. 2005. [Online]. Available: www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt. [Accessed: Mar. 2, 2010].

[3] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," In Proc. of the Intern. Symposium on Secure Software Engineering (ISSSE2006), Mar. 2006.

[4] E. M. Fayó, "Advanced SQL Injection in Oracle databases," Argeniss Information Security, Black Hat Briefings, Black Hat USA, Feb. 2005. [Online]. Available: http://www.orkspace.net/secdocs/Web/SQL%20Injection/Advanced%20SQL%20Injection%20In%20Oracle%20Databases.pdf. [Accessed: Mar. 18, 2010].

[5] "Oracle® Database PL/SQL Language Reference 11g Release 1 (11.1)," Oracle Corp., 2009. [Online]. Available: http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28370/toc.htm.[Accessed: Feb. 19, 2010].

[6] "SQL Injection Tutorial," Oracle Corp., 2009. [Online]. Available: http://stcurriculum.oracle.com/tutorial/SQLInjection/index.htm. [Accessed: Mar. 11, 2010].

[7] C. Anley, "Advanced SQL Injection in SQL Server Applications," NGS Software Ltd., United Kingdom, 2002. [Online]. Available: http://www.ngssoftware.com papers/advanced_sql_injection.pdf. [Accessed: Feb. 09, 2010].

[8] D. Litchfield, "Lateral SQL Injection: A New Class of Vulnerability in Oracle," NGS Software Ltd., United Kingdom, Feb. 2008. [Online]. Available: www.databasesecurity.com/dbsec/lateral-sqlinjection.pdf. [Accessed: Mar. 15, 2010].