Scanning and Recovery Tool for SQL Injection for Asp.Net Websites

[1]Karthik. V, [2]Venkatesh K

Department of Information Technology, SRM University, Chennai

***Abstract*— Now a days most of the web attacks comes under the structured query language i.e. SQL. This SQL helps acts as the communicator between the user and the server. SQL is just a set of queries which helps the user to get the data from the prescribed server. This SQL just acts as the backend for all the web application project. Here attacker inserts SQL characters or keywords into a SQL statement via unrestricted user input parameters to change the intended query's logic. By manipulating this data to modify the statements, an attacker can cause the application to issue arbitrary SQL commands and thereby compromise the database. To avoid this type of attacks in asp.net website we need to develop a Risk free website, calls for integrating defensive Coding practices with both vulnerability detection and runtime attack Prevention methods.**

I. INTRODUCTION

In recent years, the attacks which are being takes place in The Open Web Application Security Project (OWASP) comes under the SQL attacks. In 2011, the National Institute of Standards and Technology's National Vulnerability Database (nvd.nist.gov) reported 289 SQL injection vulnerabilities in websites, including those of IBM, Hewlett-Packard, Cisco, WordPress, and Joomla. In December 2012, SANS Institute security experts reported a major SQL injection attack (SQLIA) that affected approximately 160,000 websites using Microsoft's Internet Information Services (IIS), ASP.NET, and SQL Server frameworks.

The improper coding and improper parameters make the coding more vulnerable. This tends to the SQL injection attacks. In 2006, William Halfond, Jeremy Viegas, and Alessandro Orso2 evaluated then-available techniques and called for more precise solutions. In reviewing work during the past decade, we found that developers can effectively combat SQL injection using the right combination of state- of-the art methods. However, they must develop a better understanding of SQL injection and how to practically integrate current defences.

II. IMPROPER CODING PRACTICES

SQL is the language which is standard and used for accessing the database servers which includes MySQL, Oracle and SQL Server. It also includes some of the web programming language such as Asp.net, Java and PHP. The main thing hoe the developer's gives the insecure code is due to lack of training in their companies and also the lack of experience. Finally this tends to the insecure coding which makes the website more vulnerable.

Developers commonly rely on dynamic query building with string concatenation to construct SQL statements. During runtime, the system forms queries with inputs directly received from external sources. This method makes it possible to build different queries based on varying conditions set by users. However, as this is the cause of many SQLIVs, some developers opt to use parameterized queries or stored procedures. While these methods are more secure, their inappropriate use can still result in vulnerable code. In the PHP code examples below, name and pwd are the "varchar" type columns and id is the "integer" type column of a user database table.

Absence of checks: The most common and serious mistake developers make is using inputs in SQL statements without any checks. The following query is an example of such a dynamic SQL statement

Statement = ***"SELECT * FROM users WHERE name = '" + username + "';"***

Attackers can use tautologies to exploit this insecure practice. In this case, by supplying the value x' OR '1'='1 to the input parameter name, an attacker could access user information without a valid account because the WHERE- clause condition becomes

WHERE name = 'x' OR '1'='1' AND …";

This query compromises and which tends to true. we create regions in the image by extending the line segments

Incorrect type handling: This form of SQL injection occurs when a user-supplied field is not strongly typed or is not checked for type constraints. This could take place when a numeric field is to be used in a SQL statement, but the programmer makes no checks to validate that the user supplied input is numeric

Statement: = ***"SELECT * FROM userinfo WHERE id = " + a_variable + ";"***

It is clear from this statement that the author intended a_variable to be a number correlating to the "id" field. However, if it is in fact a string then the end-user may manipulate the statement as they choose, thereby bypassing the need for escape characters. For example, setting a_variable to drop table users will drop (delete) the "users" table from the database, since the SQL would be rendered as follows:

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

III. PROPOSED TOOL FOR DEFEATING SQL INJECTION VULNERABILITIES AND ATTACKS

Sql injection defenses

SQL injection defense methods can be broadly classified into three types: defensive coding, SQLIV detection, and SQLIA runtime prevention.

1. *SQLIV detection*

Code Analysis: You can configure Code Analysis to run before each build of a managed code project. You can set different Code Analysis properties for each Visual Studio configuration.

Vulnerability Scan: Following the discovery stage this looks for known security issues by using automated tools to match conditions with known vulnerabilities. The reported risk level is set automatically by the tool with no manual verification or interpretation by the test vendor. This can be supplemented with credential based scanning that looks to remove some common false positives by using supplied credentials to authenticate with a service. An SQLIV exists when an SQL statement does not keep statement structure and input separate. An SQL statement is vulnerable to having the logic of the statement changed by input at runtime when the application sends the structure and input of the statement together in a combined request to the database. An SQLIV is caused by dynamic SQL statement construction combined with inadequately-verified input, which allows the input to change the structure and logic of a statement. The vulnerable statement concatenates the input inputUserName with the statement structure before sending the statement to the database, which allows inputUser-Name to change the WHERE clause and the ending of the statement.Additionally, an SQL statement can contain a logical SQLIV if adeveloper creates a statement with the intent to have the structure of the statement to change based on input. A developer has to change the logic of the SQL statement and limit the range of acceptable SQL structures to remove this type of SQLIV.

2. *Prepared Statement*

Prepared statements are SQL statements that separate statement structure from statement input. Prepared statements have a static structure when they are executed and take type-specific input parameters. When prepared statements are created and the statement structure is explicitly set before runtime, the statement structure cannot be changed by input variables and the statement is mitigated from the risk posed by SQLIVs. A prepared statement is ‘‘prepared” by declaring the structure of the statement and putting bind variables, placeholders for input, in the places where SQL input goes. The SQL statement structure with the bind variables included is then sent to the database, which compiles and saves the statement structure for future execution with input variables. A prepared statement may look like this

SELECT password FROM users WHERE userName = ?

Where the question mark (?) is the bind variable. A setter method sets a bind variable as well as performs strong type checking and will nullify the effect of invalid characters, such as single quotes in the middle of a string. The setter method, set-String(index, input), sets the bind variable in the SQL structure indicated by the index to input. For example, a call to set String(1, ‘‘user1’’) would set the bind variable in the above example to ‘‘user1’’. Additionally, the setter method set Object(index, input) will call the appropriate setter method based on the object type of the input. After the SQL statement has been prepared, one setter method is used per bind variable to fill the bind variable with input. The static nature of a prepared statement’s structure is the characteristic that prevents SQLIVs. Further, since PreparedStatements,6 the Asp.net implementation of prepared statements, can be compiled once and executed multiple times, Prepared Statements are used for efficiency as well as security.Although Prepared Statements’ static structure enables Prepared Statements to avoid SQLIVs, Prepared Statements can be created which have SQLIVs if they are not developed carefully. If a developer uses input strings as part of the structure of a prepared statement, then the input changes the structure and nature of the statement before it is ‘‘prepared” and the Prepared-Statement would reflect the changes

*SELECT * FROM table name WHERE username ='’”+name+”’’*

The statement shows that the given statement contains the vulnerability where the argument passes (i.e) the ‘”+name+”’does not contains any parameterised string so that leads to the vulnerability in the code part and also this makes some attacks named as piggybag and tautology type of attacks

Detection

	FileName	LineNo	Defense Code Practise	Feasible Vulnerability
▸	AccountDe...	40	Parame...	tatulogies , piggybacked queries,Inferen...
	AccountSu...	27	Tainted ...	tatulogies , piggybacked queries,Inferen...
	AccountSu...	30	Parame...	tatulogies , piggybacked queries,Inferen...
	Default.asp...	28	Tainted ...	tatulogies , piggybacked queries,Inferen...
	Default.asp...	33	Parame...	tatulogies , piggybacked queries,Inferen...
	Deposit.as...	32	Tainted ...	tatulogies , piggybacked queries,Inferen...
	Deposit.as...	38	Parame...	tatulogies , piggybacked queries,Inferen...
	Deposit.as...	50	Parame...	tatulogies , piggybacked queries,Inferen...
	Deposit.as...	63	Parame...	tatulogies , piggybacked queries,Inferen...
	Deposit.as...	67	Parame...	tatulogies , piggybacked queries,Inferen...
	ThirdParty....	34	Parame...	tatulogies , piggybacked queries,Inferen...
	ThirdParty....	50	Tainted ...	tatulogies , piggybacked queries,Inferen...
	ThirdParty	64	Parame	tatulogies , piggybacked queries,Inferen

Fig.1 The above figure just displays the possible attacks and also the defensive code practise which we had uploaded in that tool which is done by asp.net

3. *Prepared Statement Replacement Algorithm*

The PSR-Algorithm is targeted to the environment where existing source code contains SQLIVs that need to be removed. The PSR Algorithm analyzes source code containing SQLIVs and generates a specific recommended code structure containing prepared statements.The PSR-Algorithm separates the SQL statement's input from the SQL structure in the generated code structure. The PSR Algorithm creates an additional string object for each string object used to create the SQL statement. The new string object contains the raw string data of the original string object and any identifiers found in the original string object the PSR-Algorithm identifies as SQL structure. The PSR-Algorithm creates an assistant vector for each new string object. The assistant vector is created to contain any SQL input found in the original string object. The PSR-Algorithm- generated string objects can contain other PSR-Algorithm generated string objects based on how the original string objects are used. Therefore, assistant vectors can contain other assistant vectors, creating a tree. The significance of the assistant vector tree is that it can branch based on conditionals, which makes the tree contain the proper variables for each decision path of the conditional.

4. *PSR Algorithm Implementation*

The PSR-Algorithm uses the objects involved in the SQLIV to create the prepared statement. The required objects include the Execution method, the string objects containing the SQL statement, and the Connection or Statement object. The PSR-Algorithm cannot work without these objects. The PSR-Algorithm starts separating the SQL statement structure from input by iterating through each string object used in the SQL structure and creating a new string object and a new vector object for each existing string object. The PSR-Algorithm parses each existing string object into raw string data, and identifiers. The PSR-Algorithm leaves raw string data and any guaranteed secure identifiers in the new string object as structure. A guaranteed secure identifier is an identifier the developer determines is secure through manual static analysis. The PSR-Algorithm allows all guaranteed secure identifiers to be part of the SQL structure. Of their maiming identifiers, the PSR-Algorithm identifies all non-string identifiers, assumes they are SQL input, puts them into the assistant vector, and replaces each identifier with a bind variable. The PSR-Algorithm determines if any of the remaining string identifiers have already had string and vector pairs made for them. If so, the PSR-Algorithm replaces the existing identifier with the PSR-Algorithm-created identifier and puts the assistant vector into the current assistant vector. For all of the string identifiers that the PSR-Algorithm has not converted yet, the PSR-Algorithm recursively repeats the new string and vector creation process until all string objects have associated PSR-Algorithm-generated string and vector objects. Additionally, the PSR-Algorithm recursively repeats the assignment of the new string and vector pairs for each assign of the existing string object.

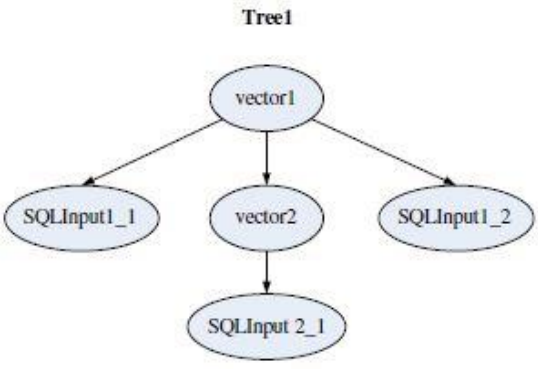

Fig. 2

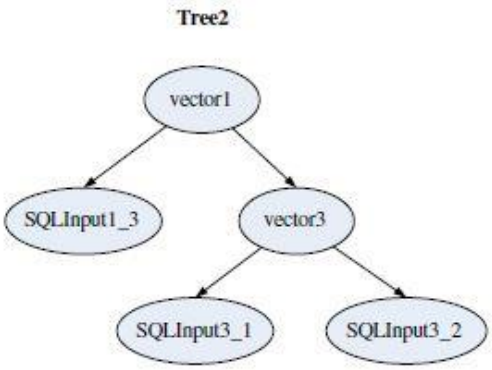

Fig.3

The Fig 2 shows an example of the type of string and vector objects the PSR-Algorithm creates. With the SQL input separated from the SQL structure via the new string and vector objects, the PSR-Algorithm creates a Prepared Statement from the new SQL structure string objects. Further, the PSR-Algorithm creates a call to the traverse Input-Tree method, which does an in-order traversal of the assistant vector tree and returns a single vector with the elements in the proper order. The PSR-Algorithm then creates a loop that goes through the single assistant vector and assigns each element to each bind variable in order. The loop assigns each element to a bind variable. The PSR-Algorithm creates a call to the execute method of the Prepared Statement and replaces the call to the Statement execute method with a call to the Prepared Statement execute method. Fig. 3 shows an example of the PSR-Algorithm preparing the Prepared Statement, setting the bind variables to the appropriate values and replacing the SQLIV with the Prepared Statement execute. Fig. 3shows the Prepared Statement created by stmt's Connection, set to the SQL structure string the bind variables set to the proper inputs, and the execution inserted into the if statement that the SQLIV was .The PSR-Algorithm inserts the traverse Input Tree method into the source code since the tree grows dynamically at runtime and needs to be traversed at runtime. The PSR-Algorithm finishes after the SQLIV execution is replaced.

5. *Model based testing*

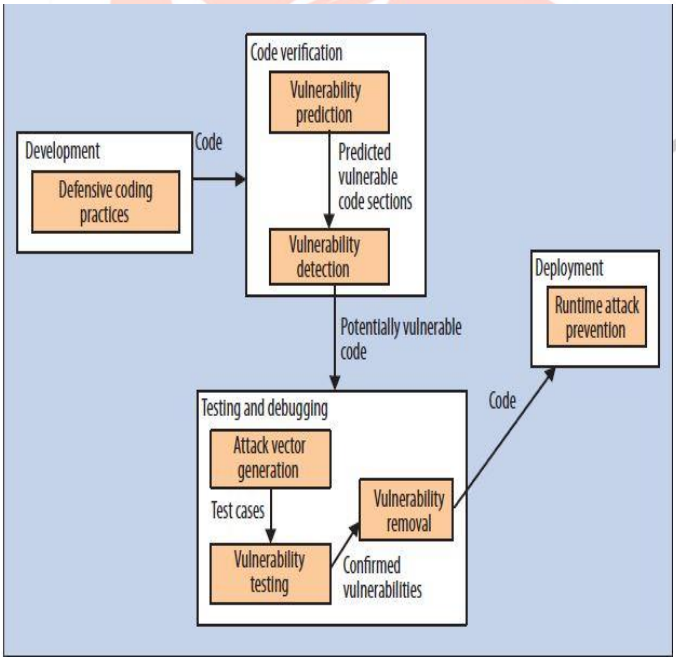

Fig.4

The Fig.4 displays the Web application developers could overcome the shortcomings of individual SQL injection methods by combining various schemes. Model-based testing approaches the system is modelled by a set of logical expressions (predicates) specifying the system's behaviour.

Constraint Logic Programming and Symbolic Execution

Constraint programming can be used to select test cases satisfying specific constraints by solving a set of constraints over a set of variables. The system is described by the means of constraints. Solving the set of constraints can be done by Boolean solvers (e.g. SAT-solvers based on the Boolean satisfiability problem) A solution found by solving the set of constraints formulas can serve as a test cases for the corresponding system. Constraint programming can be combined with symbolic execution. In this approach a system model is executed symbolically, i.e. collecting data constraints over different control paths, and then using the constraint programming method for solving the constraints and producing test cases.

6. *Test case generation*

Model checkers can also be used for test case generation. Originally model checking was developed as a technique to check if a property of a specification is valid in a model. When used for testing, a model of the system under test, and a property to test is provided to the model checker. Within the procedure of proofing, if this property is valid in the model, the model checker detects witnesses and counter examples.

7. *SQLIA runtime prevention*

Input Validation Techniques: But can prevent some vulnerability
Least Privilege: Limitations, less permissions, inflexible
Static query statement: Not good when use dynamic query
Intrusion Detection Systems (IDS): Provide little or no protection (e.g., firewalls, proxy, Gateway)

IV. CONCLUSION

Each of the three main avenues to defeat SQL injection has its own strengths and weaknesses. Defensive coding practices will ensure secure code but are time-consuming and labor-intensive. Vulnerability detection approaches can identify most if not all SQLIVs, but they will also generate many false alarms. Runtime prevention methods can prevent SQLIAs, but they require dynamic monitoring systems. The most effective strategy calls for combining all three approaches. However, this presents two major challenges. First, Web application developers need more extensive training to raise their awareness about SQL injection and to become familiar with state-of-the-art defenses. At the same time, they need sufficient time and resources to implement security measures. Too often, project managers pay less attention to security than to functional requirements. Second, researchers should implement their proposed approaches and make such implementations, along with comprehensive user manuals, available either commercially or as open source. Too many existing techniques are either not publicly available or are difficult to adopt. Readily available tools would motivate more developers to combat SQL injection. In addition, researchers should find simple ways to effectively combine existing defensive schemes to overcome the limitations of individual methods rather than focusing exclusively on novel ones.

Traditionally, SQL injection was limited to personal computing environments. However, the increasing use of smartphones, tablets, and other portable devices has extended this problem to mobile and cloud computing environments, where vulnerabilities could spread much faster and become much easier to exploit. Security researchers therefore need to address additional SQLIV-related issues arising from the greater flexibility and mobility of emerging computing platforms as well as newer programming languages such as HTML5.The outcome of this paper as follows

1. Analysis and monitoring for a solution SQL-Injection Attacks uses all types of SQLIA defenses.
2. Responds and reports immediately.
3. No false positives
4. No way can an attacker modify SQL statement.
5. Generalized to various web applications.
6. Used a set of real web applications.
7. Real attacks were generated by a real attacker.
8. It is effective, efficient, and precise.

REFERENCES

[1] C. Anley, "Advanced SQL Injection in SQL Server Applications," white paper, Next Generation Security Software Ltd., 2002; www.thomascookegypt.com/holidays/ pdfpkgs/931.pdf.

[2] W.G.J. Halfond, J. Viegas, and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," *Proc. Int'l Symp. Secure Software Eng.* (ISSSE 06), IEEE CS, 2006; www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas. orso.ISSSE06.pdf.

[3] R.A. McClure and I.H. Krüger, "SQL DOM: Compile Time Checking of Dynamic SQL Statements," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), ACM, 2005, pp. 88-96.\\

[4] S. Thomas, L. Williams, and T. Xie, "On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities," *Information and Software Technology*, Mar. 2009, pp. 589-598.

[5] Y. Shin, L. Williams, and T. Xie, *SQLUnitGen: Test Case Generation for SQL Injection Detection*, tech. report TR 2006-21, Computer Science Dept., North Carolina State Univ., 2006.

[6] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-Based SQL Injection Vulnerability Checking," *Proc. 8th Int'l Conf. Quality Software* (QSIC 08), IEEE CS, 2008, pp. 77-86.

[7] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability & Attack Injection for Web Applications," *Proc. 39th Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks* (DSN 09), IEEE, 2009, pp. 93-102.

[8] X. Fu and C.-C. Li, "A String Constraint Solver for Detecting Web Application Vulnerability," *Proc. 22nd Int'l Conf. Software Eng. and Knowledge Eng.* (SEKE 10), Knowledge Systems Institute Graduate School, 2010, pp. 535-542.

[9] A. Kiezun et al., "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), IEEE CS, 2009, pp. 199-209.

[10] N. Alshahwan and M. Harman, "Automated Web Application Testing Using Search Based Software Engineering," *Proc. 26th IEEE/ACM Int'l Conference Automated Software Eng.* (ASE 11), IEEE, 2011, pp. 3-12.

[11] K.J. Biba, *Integrity Considerations for Secure Computing Systems*, tech. report ESD-TR-76-372, Electronic Systems Division, US Air Force, 1977.
[12] V.B. Livshits and M.S. Lam, "Finding Security Vulnerabilities in Java Programs with Static Analysis," *Proc. 14th Conf. Usenix Security Symp.* (Usenix-SS 05), Usenix, 2005; http://suif.stanford.edu/papers/usenixsec05.pdf.
[13] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," *Proc. 15th Conf. Usenix Security Symp.* (Usenix-SS 06), Usenix, 2006; http://theory.stanford.edu/~aiken/publications/papers/ usenix06.pdf.
[14] G. Wassermann and Z. Su, "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 07), ACM, 2007, pp. 32-41. 15.
[15] L.K. Shar and H.B.K. Tan, "Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), IEEE, 2012, pp. 1293-1296.
[16] S.W. Boyd and A.D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," *Proc. 2nd Conf. Applied Cryptography and Network Security* (ACNS 04), LNCS 3089, Springer, 2004, pp. 292-302.
[17] K. Kemalis and T. Tzouramanis, "SQL-IDS: A Specification- Based Approach for SQL-Injection Detection," *Proc. ACM Symp. Applied Computing* (SAC 08), ACM, 2008, pp. 2153-2158.
[18] Y.-W. Huang et al., "Securing Web Application Code by Static Analysis and Runtime Protection," *Proc. 13th Int'l Conf. World Wide Web* (WWW 04), ACM, 2004, pp. 40-52.
[19] W.G.J. Halfond and A. Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks," *Proc. 3rd Int'l Workshop Dynamic Analysis* (WODA 05), ACM, 2005; www.cc.gatech.edu/~orso/ papers/halfond.orso.WODA05.pdf.
[20] K. We, M. Muthuprasanna, and S. Kothari, "Preventing SQL Injection Attacks in Stored Procedures," *Proc. Australian Software Eng. Conf.* (ASWEC 06), IEEE CS, 2006, pp. 191-198.