International Journal of Computer Science and Informatics

Volume 1 | Issue 2 Article 13

October 2011

Model Based Approach to Prevent SQL Injection Attacks on .NET Applications

Shikhar Jain
Computer Science and Engineering Deptt., National Institute of Technology, Karnataka, Surathkal, India, shikhar_jain@yahoo.co.in

Alwyn R. Pais
Computer Science and Engineering Deptt., National Institute of Technology, Karnataka, Surathkal, India, alwyn.pais@gmail.com

Available online at www.interscience.in

Model Based Approach to Prevent SQL Injection Attacks on .NET Applications

Shikhar Jain & Alwyn R. Pais
Computer Science and Engineering Deptt., National Institute of Technology, Karnataka, Surathkal, India
E-mail : shikhar_jain@yahoo.co.in, alwyn.pais@gmail.com

Abstract **-** Web applications support static and dynamic queries to access the database. Dynamic queries take input from the user and use that input to form the query. A user can give malicious input to the application which results in an incorrect query or an unauthorized query and performs vulnerable action on the database. In this paper, we presented an approach to prevent SQL injection attack (SQLIA) on .Net applications using static and dynamic analysis of the queries. The paper explains comparison of Dynamic query model and static query model in order to validate the query before sending it to the database. The result obtained proves that our designed tool has achieved prevention from SQL injection at greater extend.

Keywords— *SQL injection attack, Static Analysis, Dynamic Analysis, Code-Injecton, Validation.*

.

I. INTRODUCTION

SQL injection attacks are the most common attack on the web applications. It is among the top 10 security risks on the web applications[1].SQL injection is an attack in which attacker gives SQL tokens as an input when such input is used to form query, it changes the structure of the query and allows the attacker to get access to confidential data or modify database. SQL injection exploits confidentiality and integrity of the application.

Web applications uses database as the backend to store the information. Web applications takes user input and use those inputs to form dynamic queries, which are sent to the dataset for execution. These queries are represented as strings by the web applications. These strings are send to the database for execution. If an attacker provides SQL code as input to the application the query thus formed contains malicious code. Database will treat the query as coming from an authorized source and execute it. The malicious code will also be executed which can result from revealing critical information to compromise the database.

In this paper, we use the static-dynamic model analysis to prevent SQL injection attack. Static model define query structure at compile time while dynamic model define query structure at the run time. This is completely automated approach which is performed after complete development of the application.

Section II presents classification of SQL injection attack's, section III presents existing approaches to prevent SQL injection attacks, section IV presents our approach, section V presents how the approach is implemented and section VI concludes the paper.

II. CLASSIFICATION OF SQLIA

There are various types of SQL injection attacks performed on applications [2]. We present some of the classifications which are handled by our tool:

A. Tautologies

Tautology based attacks inject code in the conditional statements of the query such a way that they evaluate it to true. This kind of attack is used to bypass authentication and extracting data.

Example: General login page has a query like

*Select * from dept where username='john' and password='abc';*

An attacker can provide the username as "' or 1=1—", the query will become

*Select * from dept where username='' or 1=1—' and password='';*

Condition "1=1" forms the tautology and will always evaluate to true, "—"comment the password check. So the database returns all the rows in the table.

B. Illegal/Logically Incorrect Queries

In this attack an attacker provides malicious input to the system such that query formed has incorrect syntax or semantics. When such a query is executed on the database server, it generates an error message. Error messages are often descriptive and can reveal injectable parameters. They can contain confidential information

such as metadata of the database (table names, data types, column names, query structure etc.). This attack is considered as an information gathering step for further attacks.

Example: If the input for the above query is "'abc", query formed will be

*Select * from dept where username=''abc' and password='';*

Error message: *System.Data.OleDb.OleDbException: Syntax error (missing operator) in query expression 'username=''abc' and password='';'*

From the error message intruder can deduce that the type of database connectivity and the fields name in the "dept" table. This information can be used to perform further attacks on the application.

C. Union Query

In union query attack, the attacker unions malicious query with the initial query using the keyword "union". Result of malicious query is added to the result of original query.

Example: If the vulnerable input is"' union select * from dept where '1'='1", query will be-

*Select * from dept where username='' and password='' union select * from dept where '1'='1';*

Therefore above query will return all the rows in the "dept" table and allows the attacker to login into the system with the first row in the result.

D. Piggy-backed Query

Piggy-backed query attacks are possible on the databases which allow multiple queries to be contained in single string. In this attack, the intruder injects multiple queries to the original query. When the original query is sent to the database for execution, malicious queries also get executed which exploits the database.

Example: If the vulnerable input is "'; Update Into dept Values('john', 'john') –", query will be

*Select * from dept where username='' and password=''; Update Into dept Values('john', 'john') --;*

Above query will allow the attacker to insert a new login id to login into the system.

III. RELATED WORK

A number of solutions are proposed to prevent SQL injection attacks. Almost all the solution requires either the developer to check the injection parameters during development phase of the application or the application is modified to prevent SQL injection attacks.

SQLrand [3] protects from SQL injection by randomizing the SQL statement, creating instances of the language that are unpredictable to the attacker. It is implemented as database server proxy. Variable Normalization [4] extracts the basic structure of the SQL statement. SQL statement is verified against allowable list before executing. Allowable list can be defined by the developer or by the auto learning process. WAVES tool [5] identifies all the input points where SQL injection attacks are possible. It uses machine learning to test for SQL injection vulnerabilities. CANDID tool [6] mines intended query by dynamically evaluating runs over benign candidate and detects attacks by comparing it against the structure of the actual query issued. AMNESIA tool [7] combines static analysis and runtime monitoring. In static phase it builds the query model and in dynamic phase it checks the dynamic query with the static model. Our approach is also based on static and dynamic analysis. AMNESIA tool is developed to prevent SQL injection attacks on Java. It uses Java String Analyzer[8] for static analysis. Our tool is developed to prevent SQL injection attacks on .Net. We developed a prototype model of .Net string analyzer successfully which is not available till date.

IV. APPROACH

Program contains the structure of the query. Approach is to extract query structure from the programs using static analysis and at the runtime capture the dynamic query and validate it against the static query model. SQL injection attack will add tokens to the user input and hence change the query structure. If the dynamic query contains malicious code than it will not match the static query model then it will be rejected. The SQLIA prevention has four steps. The overview of SQLIA prevention used is shown in Figure 1.

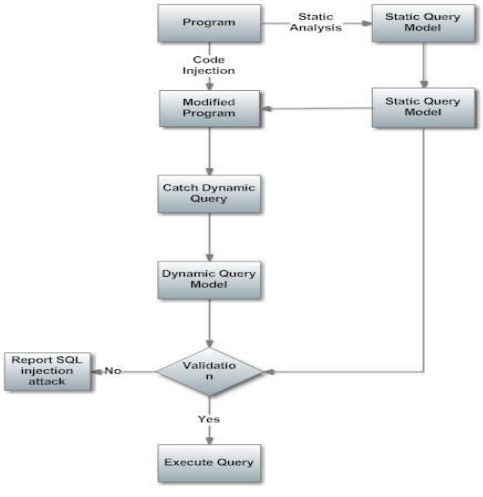

Figure 1. Overview of SQLIA prevention

A. Static Analysis

Static analysis determines the structure of the query. This step identifies the queries present in the program and generates the structure of the queries i.e. static query mode. Static query model is represented by the DFA of the query, which represents all possible values query can have.

Prototype model of string analyzer for .Net language is developed for static analysis of the program (described in next section). String analyzer generates the grammar for all the strings present in the program. We extract regular grammar for query strings only and generate the DFA for them. DFA uses SQL tokens as symbols.

Example: Query string present in the program-

```
String queryString="Select name from dept where";
if(uname=="" && pass=="")
{
    queryString+=" username='guest';";
}
else
{
        queryString+=" username='"+uname+"' password='"+pass+"';";
}
```

Static query model of the above string is shown above in Figure 2.

B. Code-Injection

In this step, original program is modified such that query string formed by including the user input is extracted before execution. Dynamic queries are sent for validation with the static query model.

C. Dynamic Analysis

Dynamic analysis takes the query formed at the runtime and form the dynamic query model for it. Tokenization of query strings into SQL tokens is dynamic query model. Query string tokenized into keywords, identifiers, special characters, strings and numbers.

Example1: Dynamic query model for genuine user where input are "smith" and "doughlas" is-

KY_SELECT ID_name KY_FROM ID_dept KY_WHERE ID_username SC_EQUALS SC_QUOTES STRING SC_QUOTES KY_AND ID_password SC_EQUALS SC_QUOTES STRING SC_QUOTES SC_SEMICOLON

Example2: Dynamic query model for malicious user where input are "' or 1=1 –" and "'" is-

KY_SELECT ID_name KY_FROM ID_dept KY_WHERE ID_username SC_EQUALS SC_QUOTES SC_QUOTES KY_OR NUMBER SC_EQUALS NUMBER SC_COMMENT KY_AND ID_password SC_EQUALS SC_QUOTES SC_QUOTES SC_SEMICOLON

D. Validation

In this step, dynamic query model is parsed against static query model. If malicious input is given to the system, then the dynamic query will not be validated and SQL injection attack will be reported.

Dynamic query model in Example1 presented in dynamic analysis section will be validated by the system as it is coming from genuine user while the dynamic query model in Example2 presented in dynamic analysis section will not be validated as it did not satisfy the static query model shown in Figure 2.

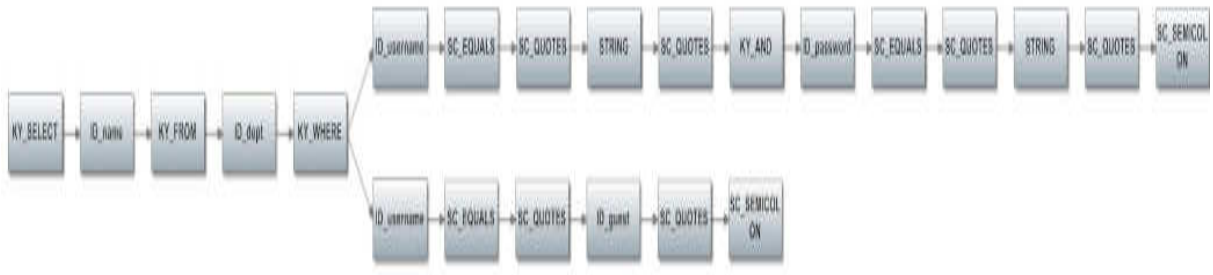

Figure 2. Static Query model

V. IMPLEMENTATION

Presented approach is implemented in C# to prevent SQL injection attacks on .Net applications. Implementation includes tools, DNSA (Dot Net String Analyzer) and SDMGV (Static Dynamic Model Generator and Validator).

A. Dot Net String Analyzer(DNSA)

- Input: Dll or exe files
- Output: Regular grammar for the program

DNSA is a prototype model of .Net String Analyzer, it takes dll or exe file as the input and convert them to MSIL (Microsoft Intermediate Language) code. We define a small set of instructions called Intermediate Instruction Set. Each MSIL instruction as defined [9] has a mapping in Intermediate instruction set. MSIL instructions are converted to intermediate instructions using the mapping. Intermediate instructions are parsed to generate the context free grammar for the given input. Grammar represents the structure of strings present in the program. Grammar is converted to regular grammar using approximation algorithm [10]. Fig. 3 shows the steps in DNSA.

B. Static Dynamic Model Generator and Validator(SDMGV)

- Input: Regular grammar from DNSA and dll or exe file
- Output: Modified dll or exe file

Figure 3. DNSA Steps

SDMGV takes output of DNSA (regular grammar of the program) and extracts regular grammar for each query string and convert it to static query model. Unique key is assigned to each static query model. Existing application is modified such that before query string is used at runtime, validate() function is called which takes the dynamic query and the static query model key as the input. Validate() function creates the dynamic query model of the query and checks dynamic query model against static query model. If validate() function returns true only then the query is allowed to be used else exception will be thrown and the query use will be skipped.

V. RESULTS

We test our tool on two web applications developed by third parties. Applications are deployed on Windows 7, IIS v6.0 Web Server and MSSQL Server. A set of injection strings are collected to attack the applications. Table 1 shows the results obtained.

Table 1. Results Obtained

Application	**Query Strings**	**Injection Attacks Performed**	**Attacks Prevented**
Book Store	74	532	532
Portal	61	438	438

Tool was able to prevent all the injection attacks performed on the web applications.

VII. CONCLUSIONS

The method presented to prevent SQL injection attack is completely automated. It is developed to prevent SQL injection attacks on applications developed in .Net language. Presented approach is based on the assumption that the program contains structure of the queries. It uses validation of static and dynamic query model for the prevention. Prototype model of .Net string analyzer is developed for static analysis phase which generates the regular grammar for the strings present in the program.

Tool can be used to prevent SQL injection attacks on existing applications as well as new applications. It doesn't depend on developer to prevent SQL injection attack, hence saves development time. Tool is tested on the sample web applications, results show that it is able to prevent and report all the SQL injection attacks performed on the web applications.

REFERENCES

[1] OWASP, O. W. (2010). "Top ten most critical web application vulnerabilities". http://www.owasp. org/ index.php\Top 10 2010-Main

[2] WG Halfond, J Viegas, A Orso , "A Classification of SQL Injection Attacks and Countermeasures," In Proceedings of the International Symposium on Secure Software Engineering, 2006.

[3] Boyd, S. W., & Keromytis, A. D. (2004). "SQLrand: Preventing SQL injection attacks," In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, (pp. 292-302).

[4] Sam M.S. N.G, "SQL Injection Protection by Variable Normalization of SQL Statement," www.securitydocs.com/library/3388, 06/17/2005.

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

[6] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, V.N. Venkatakrishnan. "CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations," In CCS '07 Proceedings of the 14th ACM conference on Computer and communications security.

[7] William G.J. Halfond, Allesandro Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL Injection Attacks," ACM, USA, 2005, pp 174-183.

[8] A. S. Christensen, A. Møller, and M. I. Schwartzbach. "Precise Analysis of String Expressions," In Proceedings of the 10th International Static Analysis Symposium, SAS 03, volume 2694 ofLNCS, pp 1–18. Springer-Verlag, June 2003.

[9] ECMA International, Standard ECMA-335 - Common Language Infrastructure CLI, Partition I-VI, ,Edition-4, June 2006.

[10] Mehryar Mohri and Mark-Jan Nederhof. Robustness in Language and Speech Technology, chapter 9: Regular Approximation of Context-Free Grammars through Transformation. Kluwer Academic Publishers, 2001.

❑❑❑