SECURING WEB APPLICATIONS FROM APPLICATION-LEVEL ATTACK

A thesis submitted
to Kent State University in
partial fulfillment of the requirements
for the degree of Master of Science

by
Amit Kumar Pandey
August, 2007
Thesis written by

Amit Kumar Pandey

B.E., Pune University, India, 2002

M.S, Kent State University, 2007

Approved by

Dr. Michael Rothestein, Advisor

Dr. Robert A. Walker, Chair, Department of Computer Science

Dr. Jerry Feezel, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................ VI

DEDICATION ................................................................................................................ VIII

ACKNOWLEDGEMENTS ........................................................................................ IX

CHAPTER 1 INTRODUCTION .................................................................................. 1

1.1 Introduction ........................................................................................................... 1

1.2 Architecture of a web application ......................................................................... 3

1.3 Web Applications and SQL injection ................................................................... 4

CHAPTER 2 AN OVERVIEW OF THE PROBLEM ............................................... 5

2.1 Motivation ............................................................................................................. 5

2.2 Research Contribution ......................................................................................... 7

2.3 Organization of thesis ........................................................................................ 8

CHAPTER 3 PROBLEM DESCRIPTION ................................................................ 9

3.1 Different cases of SQL injection ......................................................................... 9

3.1.1 Manipulating WHERE ................................................................................ 9

3.1.2 Code Injection ............................................................................................ 10

3.1.3 Exploiting INSERT .................................................................................... 11

3.2 Using error message to get database names and column .................................. 12

CHAPTER 4 RELATED WORK ............................................................................ 16
CHAPTER 5 JAVACC: PARSER GENERATOR

5.1 Introduction to lexical and parser analyzer tool: JavaCC

5.2 Lexical analyses

5.3 Lexical analyzer in JavaCC

5.4 Syntax analyzer

CHAPTER 6 IMPLEMENTATION

6.1 Architecture of security system

6.2 Algorithm used

6.3 Prefetcher

6.4 Code analyzer

6.5 Lexical analyzer

6.6 Syntax Analyzer

6.7 Generating XML from parser

6.8 Representing XML as tree structure

6.9 Detecting SQL injection

CHAPTER 7 SIMULATION

7.1 Experimental Setup

7.2 Case I

7.2.1 Using parser to generate static model

7.2.2 Generating XML at runtime

7.2.3 Comparing static and runtime model

7.3 Case II
7.4 Performance Evaluation .................................................................................. 54

CHAPTER 8 CONCLUSION ..................................................................................... 57

REFERENCES ........................................................................................................ 59
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Architecture of a typical web based system</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Lexical and syntax analyzer generated by JavaCC compiler</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Operations of lexical analyzer depicted by state machine</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Architecture for analyzing Web application and building XML</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Insertion of required code to pass the generated SQL statement to code analyzer</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>Token section</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>Production rules for SQL statements</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>Representing productions in JavaCC</td>
<td>34</td>
</tr>
<tr>
<td>9</td>
<td>Generating XML tags while parsing SQL statement</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>XML representation of SQL statement obtained using parser</td>
<td>36</td>
</tr>
<tr>
<td>11</td>
<td>Tree structure of XML obtained using DOM parser</td>
<td>37</td>
</tr>
<tr>
<td>12</td>
<td>Comparing static and runtime model to detect SQL injection</td>
<td>38</td>
</tr>
<tr>
<td>13</td>
<td>Algorithm for comparing two XML document for structural similarity</td>
<td>39</td>
</tr>
<tr>
<td>14</td>
<td>Dynamically generated SQL statement</td>
<td>40</td>
</tr>
<tr>
<td>15</td>
<td>XML generated by parser for SQL statement given in figure 7.1</td>
<td>41</td>
</tr>
<tr>
<td>16</td>
<td>XML representing SQL statement given in section 7.2.2</td>
<td>42</td>
</tr>
</tbody>
</table>
17  DOM representation for XML given in figure 7.2.................................44
18  XML DOM representation of malicious SQL statement..........................45
19  Response time of plain SQL versus Secured SQL..................................55
DEDICATION

This thesis is dedicated to my parents.
ACKNOWLEDGEMENTS

The writing this thesis has been one of the most important part of my course in computer science. I would like to express my gratitude to my advisor Dr. Michael Rothestein for his constant support and guidance.
CHAPTER 1

Introduction

The rapid growth of internet has created many services, which have become an integral part of our day to today life. Web applications are used for making reservations, paying bills, and shopping on-line. With advent of Business-to-Business (B2B) and Business-to-Consumer (B2C) interaction, it is has become a necessity that information be exchanged in a secure and accurate way. Most of the web applications contain security vulnerabilities which enable attackers to exploit them and launch attacks. As a result of the attacks confidentiality, integrity and availability of information are lost.

Information that can be read or copied by unauthorized users is called loss of confidentiality. Confidential information should be stored properly so that they cannot be disclosed. Credit card numbers, bank records, medical records, social security etc are example of this kind of information. Loss of integrity takes place when data is modified in an unexpected way. Attackers would intentionally tamper information resulting in loss of integrity of information. The loss of availability takes place when information is erased so that the legitimate user or authorized user would not be able to read it or use it. Loss of availability affects service oriented business
which depends upon data.

These attacks which are at application level, cannot be prevented using packet inspection firewalls which analyze individual IP packets for signatures or allow specific ports. What is needed is a mechanism which analyses the whole message stream. Attacks at application level differ from network layer attacks.

Application level attack exploit vulnerabilities present in web application code and limitations of protocol like HTTP. Attacks at the application level cannot be stopped by most network firewalls and antivirus software programs. A network firewall normally leaves port 80 open for web server. It is through this port that the web application communicates to the user. If the attacker is able to access applications he may launch attack which cannot be prevented by the firewall. For example, consider a user who has a legitimate account at a banking system. He connects to his account by authenticating and establishing a valid session. If the user is injecting code to access unauthorized information of other users, then the network firewall or Intrusion Detection Systems (IDSs) will not be able to stop him.

SQL injection is a very serious application level attack on web applications.
1.2 Architecture of Web application

![Diagram of web application architecture]

**Figure 1.1 Architecture of a typical web based system**

A web system consists of a web browser at the user end. The user is connected to the web application through the internet. A firewall protects the web system from intrusion and allows traffic at port 80 only. The web server receives request from the browser, processes them and passes the dynamic part to the application server, which processes server side code like JSP. All requests for database access are passed to the database server. The results are than shipped back to the web browser as HTML web pages.
1.3 Web application and SQL injection

Most of the current web applications use RDBMS (Relational Database Management Systems). Sensitive information like credit card, social security and financial records are stored in these databases. Web application allows users to view, edit or store information in RDBMS through programs written by web application programmers. Usually programmers who write these programs are unaware of technique for writing secure code. They would focus on implementing desired functionalities and would focus less on security aspects. This results in vulnerabilities in web applications. Vulnerabilities allow attacker to target these web application and obtain valuable information. Attackers would send SQL (Structured Query language) to interact with RDBMS servers or modify existing SQL to retrieve unauthorized information without any authentication. The risk is higher if the application is open source or if the attacker is able to gain source code through other means. In this case he can analyze the code to find out the vulnerabilities
CHAPTER 2

Overview of Problem

Web applications often use data submitted by a client to construct SQL queries. The communication between the programming language and the database commonly takes place through a call level interface such as JDBC or ODBC. Call level interfaces allows full power of SQL by accepting dynamically generated SQL statements. User input is concatenated to a predefined SQL statement and is passed by CLI to the database for query. However, constructing queries by concatenating user input with SQL statements leads to vulnerability where the user can execute arbitrary SQL against the database. The work described here tries to prevent SQL injection by using an SQL parser written in java. In addition to that we have developed a data type checking system which prevents data type mismatch in dynamically generated SQL queries. An implementation of this system is described and is shown how it handles most common SQL injection and data type mismatches.

2.1 Motivation

Cross site scripting, buffer overflow, content spoofing and session fixation SQL injection are some of the application layer attacks targeting most of the current web application. Among all this attacks cross site scripting and SQL injection are very common. SQL injection is considered a severe of attack affecting
confidentiality, integrity and availability of information. SQL injection represents one of the greatest threats for the following reasons:

- Nearly all websites have some sort of database at the back end.
- Databases often house sensitive data such as credit card numbers, social security numbers and other personal data that attackers would normally target.
- Many web applications are written by non-engineers who have little knowledge of writing them in a secure way, combined with fact that web development has rapid development cycles resulting in code which is vulnerable to SQL injection. This is further compounded by many texts available in market that programmers use with insecure SQL syntax.

Many web servers return an incorrect syntax error and part of the SQL statement that was sent to database server by the middle tier for execution. This is used effectively by attacker to generate errors by trying various input combinations and get the SQL statement in the error message. Using standard SQL errors attackers are able to get knowledge of the underlying database, tables, data type of columns and query structure. With knowledge of all these things, the attacker is able to formulate queries and attack the website to get required information.

One way to solve this problem is to improve programming practices like escaping single quotes, limiting the input character length and disabling error message generated by database server. Despite all these suggestion vulnerabilities continue to come up in web applications. Another way to minimize this problem is to use the PREPARE statement feature which is supported by many databases. Prepared
statements are precompiled SQL statements with one or more parameters. Prepared statement is generally immune to SQL injection as the database will use the value of the parameter exclusively and would not interpret the contents of the variable in any other way.

However they can be used only when the same query is to be executed many times. When the queries are dynamically generated (like if they are constructed based upon what user selects from several options), a prepared statement cannot be used. Other alternatives like SQLJ do perform static syntax and type checking but they work only with static SQL statements. Also SQLJ is not very popular and hardly used.

2.2 Research Contributions

An Architecture which prevents SQL injection by static and dynamic analysis of SQL string is presented. To implement analysis of the SQL string we wrote a top down parser using javaCC. We present a simulation of the algorithm on two different cases. The applications are written in java technology with MS SQL server as a database. The algorithm was able to detect the SQL injection in both cases.

The approach use an XML document to store the static model of the SQL statement. The use of XML allows flexibility in deploying our web security system. It makes it possible for the web developer to specify acceptable SQL string by writing their own XML document to represent the static model. Though we tested our system
for web applications written in java technology, the approach can easily be adapted for web applications written in other languages.

In addition to preventing SQL injection the architecture prevents data type mismatch in dynamically generated SQL query strings. Compared to other approaches like prepared statement, our approach works well with legacy code, does not require changes in application code and helps achieve utilize full power of CLI without sacrificing the security aspect.

2.3 Organization of the thesis

This work is organized as follows. CHAPTER 3 provides description of the different cases of SQL injection attack. It also describes how attackers are able to use the data type error returned by the database server for formulating their query for retrieving unauthorized information. CHAPTER 4 gives a survey on the related work done in the area of application layer attack, specifically SQL injection attack. CHAPTER 5 gives introduction to the parser generator tool: javaCC. CHAPTER 6 describes the various components and algorithm of our security system which we had designed and implemented to prevent SQL injection attack. CHAPTER 7 explains working of our security system for the two different cases of SQL injection attack; it also gives performance evaluation of our system. CHAPTER 8 presents our conclusions and future work.
CHAPTER 3

Problems Description

Normally web applications use input provided by the user for querying the databases. The user input is used further in generating an SQL statement which is executed at RDBMS. SQL injection is trying to give input through web application's user interface that would return malicious user sensitive information. It is also possible to edit or modify that information.

3.1 Different cases of SQL injection:

3.1.1 Manipulating WHERE:

In this case attackers manipulate SQL statements by changing the where clause of the SQL statement to get different results than a normal SQL statement. The attacker would manipulate where clause to put a tautology and disable password clause using a comment. "-- " is used to indicate comments in SQL. The attacker can use the comment symbol to truncate the query; anything that comes after the "--" will be treated as comment and will not be executed by the database server.
E.g.: consider a query

```
select * from tablename where username='a' and password = 'b', where 'a' and 'b' are
the username and the password respectively entered by the user to view his account
information.
```

The attacker enters ' OR 1=1 -- in username field and leaves the password field
blank. The query would become:

```
Select * from tablename where username=' ' OR 1 =1 -- and password = ''
```

The single quote entered by the user disables the opening quote and OR 1=1, which is
a tautology, satisfies Where clause for all records. Hence, when this query is executed
it will return information about all the users in the table.

### 3.1.2 Code injection

This type of attack is possible only if the database supports multiple SQL
statements per database request. MS SQL server and postgrSQL support it but
Oracle does not.

Consider a Student directory Website which prompts a user to enter the surname of a
student to search for by means of a form-box called searchName.
On the server-side this search string (stored in the variable LastName) is used to build an SQL query. This may involve code such as:

```
query = "SELECT forename,tel,fax,email FROM personal WHERE surname=`LastName`; ";
```

However, if the user enters the following text into the surname form box:

`; SELECT password,tel,fax,email FROM personal WHERE surname=`Jones

then the value of variable, query will become:

```
SELECT forenames,tel,fax,email FROM personal
WHERE surname=``; 
SELECT password,tel,fax,email FROM personal
WHERE surname=`Jones`; 
```

When executed on some SQL databases, this will result in Jones's password being returned instead of his forename.

### 3.1.3 Exploiting INSERT

Normally an insert statement looks like this:

```
Insert into TableName Values ('ValueOne', 'valueTwo', 'valueThree');
```

Consider a webpage which allows user to edit his information or webpage where user fills in their information.

The fields are user name: Name, user email : Email and user Phone number: Phone
If attacker puts this information in these three fields as

Name: ‘ + (SELECT TOP 1 Fieldname from TableName) + ‘

Email: abc@xyz.com

Phone: 33042364

The resulting SQL statement looks like this

INSERT INTO tableName values (‘ ‘ + (SELECT TOP 1 Fieldname FROM tableName ) + ‘’, ‘abc@xyz.com’, ‘33042364’)

In this case the first value in the FieldName will be displayed in place of the user name.

3.2 Using error messages to get database name and column

In a secured web application, if the database generates some error then the application would handle those errors and would not display it to the user. However, due to default behavior of ASP/JSP in most web applications database errors are displayed to the user. This behavior is effectively used by the attacker to generate error messages from the database server which would use help him to get the structure of database. He can than formulate queries using that information to retrieve unauthorized information.
Let’s assume that there is a Web page where a user needs to specify this username and password so that they can log into the application. The web application has default behavior, so if there is any error is generated by the database server it is shown to the user. The page uses the following SQL statement to verify the users credentials in the database.

```
Select * from users where username = ' " + u_name + " ' and password = ' " + u_password + " ' ; u_name and u_password variable stores the value of username and password submitted by the user.
```

First, the attacker would want to get the names of the tables that the query operates on, and the names of the columns. To do this, the attacker uses the 'having' clause of the 'select' statement:

```
u_name: ' having 1=1--
```

This provokes the following error:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'users.id' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.
```

The error message reveals the table name (users) and column name (id) of the first column in the query.
To retrieve the name of second column the attacker would put first column into the 'group by' clause, as follows:

```
u_name:  group by users.id having 1=1 --
```

The error produced generated by the database server would be:

Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'users.username' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

This error reveals the name of the second column username i.e. username. The process can be repeated to retrieve the name of all columns. Once the attacker gets all column information he can get all rows from table by giving following input:

```
u_name:  group by users.id, users.username, users.password, users.ssn having 1=1—
```

This input will not generate any error and the attacker would get all the rows from 'users' database. The attacker is able to do that since he able to determine the table and columns name.

**Determining the data type of column**

Attacker can determine the data types of each column using 'type conversion' error message. For example if attacker gives following input in the username field:
u_name: ' union select sum(users.username) from users —

It will generate this error message:
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]The sum or average aggregate operation cannot take a varchar data type as an argument..

This error message indicates that data type of username is 'varchar'. The attacker can repeat same process to determine the data type of other columns. Once he knows the data type of all columns he can create well formed 'insert' query by giving following input:

u_name: ' ; insert into users values(123, 'hacker', 'password' , 204890729 );
CHAPTER 4

Related Work

The prevailing method of preventing SQL injection attack works by limiting the length of user input, or by filtering the user input for bad strings which are required for attacks. Commercial product like AppScan[] and WebInspect[] uses testing techniques for finding vulnerabilities in existing code.

Huang, Tsai et al. [2] use fault injection, behavior monitoring and black box testing to identify vulnerabilities. They typically identify data entry points in application, construct potentially valid input field and inject malicious input patterns on input field. Vulnerabilities are identified by analyzing HTTP replies and monitoring browser's behavior. But like all testing techniques it cannot provide complete protection from patterns which are very different from existing one, as the testing uses existing malicious patterns to test application. Also, though testing can help in finding vulnerabilities it is not a complete solution when it comes to protecting application from attack.

Boyd, Keromytis [4] proposed SQLrand which uses instruction set randomization of SQL statement to check for SQL injection attack. It uses a proxy to append key to SQL keyword. A de-randomizing proxy then converts the randomized query to proper SQL queries for the database. The key is not known to
the attacker, so the code injected by attacker is treated as undefined keywords and expressions which cause runtime exceptions and query is not sent to database. The disadvantage of this system is its complex configuration and the security of the key. If the key is exposed, attacker can formulate queries for successful attack.

Scott and Sharp [3] developed application level firewalls which allow developers of web application to specify constraints on inputs. The constraints are compiled and than run in firewall which analyzes HTTP requests and responses and enforces those constraints. Malicious inputs are prevented from reaching web server. However their methodology requires the developer of the web application to specify the constraints based on their application, which makes it prone to false positives and negatives.

Huang et al. [4] developed a tool called WebSSARI which uses a lattice based static analysis algorithm derived from the type system to identify vulnerabilities in PHP web application. WebSSARI used coarse-grain approach i.e., any data derived from tainted input is fully tainted. WebSSARI inserts a runtime guard that filters dangerous content from tainted values before they are passed for execution.

Tuong, Salvatore et al. [6] used precise tainting of data. Unlike Huang et al. where everything derived from tainted input was considered tainted, this paper focused on precisely tracking taintedness within data values using program language semantics and using context dependent checking. To prevent SQL injection attacks, they tokenize query strings and preserve taint making. To check for attack they check
if operator symbol is marked as tainted or if identifier is tainted and keyword like UNION, DROP, WHERE, OR, AND etc.

Halfond and Orso [4] used string analysis technique [8, 9] to build a conservative model of queries. At runtime they checked if the model complies with statically generated model. Their static model is built by constructing a Non-Deterministic Finite Automaton (NDFA) representing the query at compile time. At run time they check if the NDFA accepts the query. However their methodology does not work for broad range of queries.

Su, Wassermann [10] presented a formal definition of command injection attacks in web applications and developed run time checking algorithms to command injection attack. Their methodology is similar to ours, they check syntactic change parse tree of queries to check command injection attack. However unlike our methodology their system is not flexible enough to let the developer specify constraints.

McClure and Kruger developed a new API which lets a user construct dynamic SQL statements in a way that is free from type errors and is free from SQL injection attacks, but, their approach cannot be used for existing applications. It can be used for new applications where instead of constructing dynamic SQL by concatenating strings; they are produced by calling APIs for each part of SQL.

Gould, Su and Devanbu [10] used static analysis technique to analyze all possible string for SQL strings to identify type mismatch errors. However doing only static checking makes it ineffective against SQL injection attacks.
Joshi [15] analyzed how the attacker manipulates SQL statement to launch attack; they have also shown how type mismatches can be used to identify the database table structure. They suggested different ways to mitigate problems like using stored procedures, prepared statements etc. But their suggestion does not work for existing code.
5.1 Introduction to the lexical and parser analyzer tool: JavaCC

The Java compiler compiler (javaCC) is most the popular lexical analyzer and parser generator for use with java applications. A parser generator is a tool that reads a grammar specification and converts it to a program that can recognize matches to the grammar. In addition to the parser generator itself, javaCC provides other standard capabilities related to parser generation such as tree building, actions, debugging etc.

Parsers and lexical analyzers are software components for dealing with input of character sequences. Compilers and interpreters incorporate lexical analyzers and parsers to decipher files containing programs; however lexical analyzers and parsers can also be used in a wide variety of other applications.
Figure 5.1 Lexical and syntax analyzer generated by JavaCC compiler

JavaCC source file contains specification for generating lexical and syntax analyzer. The specification is written in file with .jj extension which is compiled by JavaCC compiler to generate seven Java classes. Each class has its own file. The seven files are:

- **TokenManagerError**: It is a simple error class which is used to detect errors by the lexical analyzer. It is a subclass of Throwable.

- **ParseException**: It is another error class which is used by the parser for detecting errors. It is a subclass of Exception.

- **sqlParserTokens**: It is a class representing tokens. Each Token object has an integer field kind that represents the kind of the token and a String field image, which represents the sequence of characters from the input file that the token represents.
• **sqlParserTokenManager**: It is the lexical analyzer produced by JavaCC.

• **sqlParser**: It is the parser.

• **SimpleCharStream**: It is an adapter class that delivers characters to the lexical analyzer.

• **sqlParserConstants**: It is an interface that defines a number of classes used in both the lexical analyzer and the parser.

5.2 Lexical analyses

So what are lexical analysers and parsers? Lexical analysers can break a sequence of characters into subsequences called *tokens* and it also classifies the tokens. Consider a short program in the C programming language.

```c
int main() {
    return 0;
}
```

The lexical analyzer of a C compiler would break this into the following sequence of tokens

```
“int”, “ ”, “main”, “(”, “)”
“ ”, “{”, “\n”, “\t”, “return”
“ ”, “0”, “ “, “;”, “\n”,
“}”, “\n”, “”
```

The lexical analyzer also identifies the kind of each token. The token of kind EOF represents the end of the original file. The sequence of tokens is then passed on to the parser. In the case of C, the parser does not need all the tokens. In our example, those white space is not passed to parser.

5.3 Lexical analyzer in JavaCC

"TokenManager" is a lexical analyzer which is generated by the JavaCC compiler from source file. The token manager reads in a sequence of characters and produces a sequence of objects called "tokens" according to regular expressions specified in the source file. The regular expression kind specifies what to do when a regular expression has been successfully matched. There are four kinds:

- **TOKEN**: Token manager creates token for each regular expression which is then returned to the parser.
- **SPECIAL_TOKEN**: They are special tokens that do not participate in parsing. They are useful in processing special lexical entities such as comments. The matched token is saved to be returned along with the next TOKEN that is matched.
- **SKIP**: They are simply skipped or ignored (after executing any lexical action) by token manager.
- **MORE**: Matched string is stored in buffer until next Token matches. Then all the matches in buffer are concatenated together to form one Token and passed on to the parser.
The TokenManager is a lexical state machine that moves between different lexical states to identify a token and its kind. The generated token manager is at any moment in one of these lexical states. When the token manager is initialized, it starts off in the DEFAULT state, by default. Depending upon the input character it changes lexical state. For example if the input is character 'A' it makes a transition to state 'A'. If the next input character is again 'A' the analyzer stays in the same state. In case of other inputs like 'B' or 'C' it moves to corresponding state. If the input is any other character except these three the TokenManager generates lexical error. This operation is depicted in figure 5.2.

Figure 5.2 Operations of lexical analyzer depicted by state machine
5.4 Syntax analyzer

The syntax analyzer in JavaCC is a recursive-descent LL(k) parser. It allows use of multiple token lookahead. This type of parser uses k number of lookahead tokens to generate a set of mutually exclusive productions, which recognize the language being parsed by the parser. By default, JavaCC’s syntax analyzer sets k to 1, but developers can override the number of lookahead tokens to any arbitrary number to match productions correctly.
CHAPTER 6

Implementation

6.1 Architecture of security system

Figure 6.1 Architecture for analyzing Web application and building XML representation of SQL statements
6.2 Algorithm used

Our approach uses static and runtime analysis of dynamically generated SQL to prevent SQL injection. For performing static and runtime analysis we had designed an architecture which is depicted in figure 6.1

Following steps are involved in preventing SQL injection:

1. A Pre-Fetcher fetches web pages from web application, scans it to determine where a SQL statement is issued, and inserts some code which sends dynamically generated SQL to code analyzer.

2. Code analyzer analyzes SQL statement using a lexer and parser which is generated by JavaCC from SQL grammar represented using EBNF (Extended-Backus-Naur Form).

3. The output of parser is a XML document which represents the SQL statement using XML tags.

4. The XML document can be generated by providing input to web form by those deploying the web application or those testing the web application. The XML document represents the static model. Static model would be stored so that they can be compared with dynamic model.

5. When web application is deployed, same process would be followed to generate the XML document for SQL statement which would be generated when user of web
application would provide input. We are calling this XML document the dynamic model.

6. The dynamic model is compared with the static model by converting these XML document into a tree structure using the XML DOM API. XML DOM represents XML as a tree structure so that they can be navigated and used just like a tree data structure.

7. The trees for static and dynamic model are compared using breadth first search algorithm.

8. If the trees have same structure than there is no SQL injection, however difference in structure indicate a possible SQL injection.

9. The result of comparison is sent back to web application, if there is SQL injection the application does not execute SQL statement.

The detail working of different components involved in analyzing and comparing trees are explained with two different examples.

6.3 Prefetcher:

Pre-fetcher fetches web pages from web applications and identifies those places where SQL statements are sent to the database server for execution. For each such place it will insert required code which retrieves the SQL statement from web page and passes it to the code analyzer for analysis. It also inserts statement which will prevent the SQL statement from getting executed in case there is SQL injection. The second kind of statements are inserted after building static models for SQL statements present in web application.
dbconn = DriverManager.getConnection("jdbc:odbc:BookShopDb");
Statement st = dbconn.createStatement();
String sql = "SELECT * FROM logininfo " + "WHERE username = " + uname + ""
+ " AND password = " + pword + """ + ";
analyzer.setSql(sql);
if(analyzer.getResult() = true)
ResultSet results = st.executeQuery(sql);

Figure 6.2  Insertion of code to pass the generated SQL statement to code analyzer.

The above servlet code creates dynamic SQL from user input which are retrieved from web form. The SQL statement is then sent to database for execution using JDBC technology.

The code analyzer scans the code and inserts two statements. The first one: "analyzer.setSql(sql)" passes the SQL string to analyzer which parses it and generates the XML document. Static model can be constructed by passing the normal expected kind of value in the form field. This is done before deploying web application.
During deployment, the second kind statement is inserted. The second statement: "if(analyzer.getresult() = true)" calls the analyzer to check the result of comparing static model and dynamic model. If there is SQL injection the analyzer returns false. In that case we don't execute the SQL statement since there is a possibility of SQL injection by the user.

6.4 Code analyzer

It consists of lexer and parser which is generated by JavaCC using an SQL grammar specified using EBNF. The lexical analyzer recieves the SQL string from Pre-Fetcher unit. The lexical analyzer (token Manager) reads the SQL string and groups characters into tokens and passes it on to the syntax analyzer. Syntax analyzer checks the syntax of the SQL string and generates XML document which represents the dynamically generated SQL statements.

The main compilation unit in a JavaCC grammar file is enclosed between PARSER_BEGIN(sqlpar) and PARSER_END(sqlpar), where sqlpar is used as the prefix for all generated Java classes. The parser code that JavaCC generates is inserted immediately before the closing brace of the main compilation unit.

6.5 Lexical analyzer

A lexical analyzer consists of a list of terminal symbols, including a SKIP region indicating those symbols which the token manager has to ignore and not pass
to the parser.

```plaintext
SKIP :
{
    " 
    "\t"
    "\n"
    "\r"
}

TOKEN :
{
    <K_WHERE: "WHERE">
    <K_HAVING: "HAVING">
    <K_AND: "AND">
    <K_SELECT: "SELECT">
    <K_FROM: "FROM">
    <K_UPDATE: "UPDATE">
    <K_OR: "OR">
    <K_NOT: "NOT">
    <K_SET: "SET">
    <K_INSERT: "INSERT">
    <K INTO: "INTO">
    <K_VALUES: "VALUES">
}

TOKEN :
{
    < ID: ["a"-"z", "A"-"Z", "_"] ( ["a"-"z", "A"-"Z", "_", "0"-"9"] )* >
    < NUM: ["0"-"9"] + >
    <K_COMP: "<" | ">" | "=" | "<<=" | ">=" | "!="">
}
```

**Figure 6.3 Token section**

The first line in the token section indicates that space characters are tokens but that they need to be ignored and not passed to the parser. New line characters ("\n") and carriage return ("\t") are also ignored by the lexical analyzer by specifying them in the SKIP section. The subsequent sections specify the terminal symbols i.e. SQL keywords. For example the first line in the second section tells JavaCC that "WHERE" is a token and gives a symbolic name to that token: K_WHERE. Similarly
other SQL keywords are specified and given a symbolic name. Using option
IGNORE_CASE in JavaCC we disabled case sensitivity for SQL keywords. A
powerful feature of JavaCC is the possibility to specify fairly complex regular
expressions. The third section declares regular expression for identifiers and numbers
and declares all comparison operators as token K_COMP.

6.6 Syntax Analyzer

<table>
<thead>
<tr>
<th><strong>Input</strong></th>
<th>--&gt;</th>
<th>Sql [Sql] EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sql</strong></td>
<td>--&gt;</td>
<td>SELECT selectstm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPDATE updatestm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INSERT insertstm</td>
</tr>
<tr>
<td><strong>Selectstm</strong></td>
<td>--&gt;</td>
<td>( *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ K_WHERE wh_condition ]</td>
</tr>
<tr>
<td><strong>Updatestm</strong></td>
<td>--&gt;</td>
<td>ID SET ID COMP value wh_condition</td>
</tr>
<tr>
<td><strong>Insertstm</strong></td>
<td>--&gt;</td>
<td>ID &quot;(&quot; identlist () &quot;)&quot; VALUES &quot;(&quot; value ()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(&quot;,&quot; value () )* &quot;)&quot; &quot;&quot;,&quot;</td>
</tr>
<tr>
<td><strong>Identlist</strong></td>
<td>--&gt;</td>
<td>ID (&quot;,&quot; ID&gt;*</td>
</tr>
<tr>
<td><strong>wh_condition</strong></td>
<td>--&gt;</td>
<td>term restexpr</td>
</tr>
<tr>
<td><strong>restexpr</strong></td>
<td>--&gt;</td>
<td>OR term restexpr</td>
</tr>
<tr>
<td><strong>term</strong></td>
<td>--&gt;</td>
<td>NOT cond</td>
</tr>
<tr>
<td><strong>cond</strong></td>
<td>--&gt;</td>
<td>value comp value [&quot;--&quot;]</td>
</tr>
<tr>
<td><strong>value</strong></td>
<td>--&gt;</td>
<td>NUM</td>
</tr>
</tbody>
</table>

Figure 6.4 Production rules for SQL statements

A syntax analyzer is generated by javaCC using a sequence of productions specified
using Backus-Naur Form. The productions which were used are listed in figure 6.4
The left hand side of production is a non-terminal symbol which is represented as a method declaration in Java parser code. The right hand side of the production becomes part of that method body, and helps the parser in making choices.

```java
void identlist( ) :
{ Token t,t1 ;}
{
 t=<ID>
{...
}"
 t1=<ID>
{
 try{
 ....
 }catch(ParseException e)
 {System.out.println("Error in identifier list" )
 }
 }*
}
```

**Figure 6.5 Representing productions in JavaCC**

Parsing errors are handled using try and catch statements. Parsing errors are reported using a catch block.

### 6.7 Generating XML from parser

The XML representing an SQL statement is generated by writing an XML tag in a temporary file (static.txt) while parsing the SQL statement. For example as shown in Figure 6.6 when a parser reads "Select" in the SQL statement it matches with
K_SELECT token. This results in code inside braces following K_SELECT to be executed. Here in this example it causes a <SELECT> start tag to be written to file static.txt.

```java
<K_SELECT>
{
    try{
        FileWriter fw = new FileWriter("static.txt",true);
        PrintWriter pw= new PrintWriter(fw);
        pw.println("<SELECT>");
        pw.flush();
    } catch(IOException e){ }
}

selectstm()
{
    try{
        FileWriter fw = new FileWriter("static.txt",true);
        PrintWriter pw= new PrintWriter(fw);
        pw.println("</SELECT>");
        pw.flush();
    } catch(IOException e) { }
}
```

**Figure 6.6 Generating XML tags while parsing SQL statement**

After process production 'selectstm' (Details of which is given in Figure 6.4) it writes the appropriate tags in static.txt. Lastly as seen from the above figure the closing tag of select statement (</SELECT>) is written in the static.txt file.

So, when the parser encounters a SQL statement like this:

```
SELECT info from users WHERE login = 'doe' AND pass='xyz' ;
```

It parses it according to the production given in figure 6.4 and converts it into an XML representation.
The above mentioned SQL statement is converted into XML document given in figure 6.7

```xml
<INPUT>
<SQL>
  <SELECT>
    <ID>info</ID>
    <FROM>
      <ID>users</ID>
    </FROM>
    <WHERE>
      <ID>login </ID>
      <COMP> = </COMP>
      <ID>doe</ID>
      <AND/>
      <ID>pass</ID>
      <COMP> = </COMP>
      <ID>xyz</ID>
    </WHERE>
  </SELECT>
</SQL>
</INPUT>
```

Figure 6.7 XML representation of above SQL statement obtained using parser.

6.8 Representing XML as a tree structure

SQL injection is detected by comparing the XML for the SQL statement obtained during testing with the XML for the SQL statement obtained at runtime due to the input provided by the user. For comparing the XML document, they need to be converted into their tree representation. Tree representation is obtained by a DOM parser API provided java. It parses an entire XML document and constructs a complete in-memory representation of the document as tree structure. There are many
different types of tree nodes, representing the type of data found in an XML document. The most important node types are:

- element nodes which may have attributes
- text nodes representing the text found between the start and end tags of a document element.

The tree structure of XML shown in figure 6.7 obtained using XML DOM parser is shown in figure 6.8.

![Diagram of XML tree structure](image)

**Figure 6.8** Tree structure of XML (shown in figure 6.7) obtained using DOM parser
6.9 Detecting SQL injection

SQL injection is detected by comparing XML, representing static model, with XML representing runtime model. For comparison breadth first search algorithm is used. The result of comparison is a Boolean value indicating if SQL statement is malicious. False value indicates that SQL statement is malformed hence could be possible SQL injection attack and prevents SQL statement from being executed whereas a true value allows execution of SQL statement.

Architecture for comparing two XML document is depicted in figure 6.9 and algorithm used for comparing two documents is depicted in figure 6.10.

![Diagram comparing static and runtime model to detect SQL injection](image)

**Figure 6.9 Comparing static and runtime model to detect SQL injection**

Comparison between two documents is carried out by breadth first search algorithm. The algorithm is given in figure 6.10. Input to algorithm are XML document representing static model and XML document representing runtime model.
Both are converted into their respective DOM representation. The algorithm starts from root node of both XML DOM and computes number of children node for both root nodes. It then compares the number of children of both node to see if they are same. If the number of children are not same than it returns false value indicating that structure of both XML node is not same, which might be due to manipulation of SQL statement by attacker. If the number of children is same, than it picks child node of root node and repeats same process using breadth first search algorithm.

<table>
<thead>
<tr>
<th>Input:</th>
<th>Two XML document</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>Boolean value indicating if the tree represented by XML matches or not</td>
</tr>
</tbody>
</table>

Create Document object model (DOM) for both static and dynamic model
Create two queues one for storing nodes of static model tree, another for dynamic model tree.
Read first node of two XML trees
While (1)
  Compute number of children of two nodes, node1 and node2
  If the number of children of both nodes are different
    return false
  else
    for all children of node1 of static tree
      insert in first queue
    for all children of node2 of dynamic tree
      insert in second queue
    if both queues are empty
      return true
    else
      take out node, node1 from first queue
      take out node, node2 from second queue

**Figure 6.10 Algorithm for comparing two XML document for structural similarity**
CHAPTER 7
Simulation

7.1 Experiment Setup

For testing purpose we wrote a simple J2EE application with SQL server 2000 as database server. The home page of application is JSP page which requires user to provide user name and password to view his account information. When user provides his username and password, the information is sent to middle tier via HTTP GET protocol. Middle tier embeds user name and password in a dynamic query and sends it to database server for execution. The output of database server is formatted and sent back to front end, where user can view his account information. The query, which is being used is dynamically generated in middle tier, is being given figure 7.1

```
"SELECT * FROM info " + "WHERE username = " + uname + ""
+ " AND password = " + pword + ";"
```

Figure 7.1 Dynamically generated SQL statement

Where 'uname, and 'pword' contains user name and password provided by user to view his account information.
When we tested the application by providing string ' OR 1=1 -- in the username field and leaving the password field blank, the application in the absence of server side validation returned information about all users similar to what we discussed in problem description chapter.

7.2 Case I

In case I we examine how our architecture analyzes SQL statement and prevents 'manipulating WHERE' type of attack which we discussed in problem description chapter.

7.2.1 Using parser to generate static model

The above mentioned SQL statement is parsed to generate XML document.

The XML document which is being generated is shown in figure 7.2

```
<INPUT>
  <SQL>
    <SELECT>
      <ID>info</ID>
    </SELECT>
  </SQL>
</INPUT>
```

Figure 7.2 XML generated by parser for SQL statement given in figure 7.1
Here the root node is 'Input', 'select' node represents select statement of SQL. 'ID' identifies identifier in statement. COMP represents comparison operator and 'AND' represents AND clause of SQL statement.

7.2.2 Generating XML representing SQL statement at runtime

When attacker puts ' OR 1=1 -- in the username field and enters random value in password following SQL statement is generated:

SELECT * from info where username = " OR 1=1 -- and password = 'b';

When it is parsed by parser the XML which is generated is given in figure 7.3

![XML representation of SQL statement](image)

Figure 7.3 XML representing SQL statement given in section 7.2.2
The parser identifies OR 1=1 tautology and places them inside 'where' node, 

Similarly comment is also identified and placed inside COMMENT tag.

7.2.3 Comparing static and runtime model

XML representing static model is compared with XML representing runtime model. For comparison algorithm given in figure 7.10 is used. For comparison they are first converted into their DOM representation. In figure 7.4 we have given DOM format for XML given in figure 7.2.
Figure 7.4 DOM representation for XML given in figure 7.2

Similarly XML representing runtime model of malicious SQL statement is converted into DOM format by same parser. The resulting structure is shown in figure 7.5. The tree structure obtained after converting XML to DOM representation helps us in applying breadth first search algorithm for comparing static with runtime model.
DOM trees of figure 7.4 and figure 7.5 are compared using breadth first search algorithm implemented using queue.

1) Read root node of static XML DOM tree in queue1 and root node of dynamic XML DOM tree in queue2
2) Compute number of child node for both nodes

   Number of child node of "input" node stored in queue1 = 1

   Number of child node of "input" node stored in queue2 = 1

3) Push child nodes of both "input" nodes in their respective queue

4) Pop element from both queue
4) Compute number of child node for both nodes
   Number of child node of "SQL" node stored in queue1 = 1
   Number of child node of "SQL" node stored in queue2 = 1

5) Push child nodes of both "SQL" nodes in their respective queue
6) Pop element from both queue

7) Compute number of child node for both nodes

Number of child node of "SELECT" node stored in queue1 = 1

Number of child node of "SELECT" node stored in queue1 = 1

8) Push child nodes of both "SELECT" nodes in their respective queue

The process continues, until "WHERE" nodes are at the front of queues, at that point, the number of children node of WHERE node located in queue1 would be
7 and the number of children node of WHERE node located in queue2 would be 12 (due to the presence of tautology: OR 1=1 and comment: "--"). The matching algorithm would return false which tell that there is difference between static and runtime model of SQL and hence a possible SQL injection.

7.3 Case II

In case II we examine second type of SQL injection attack, i.e., code injection attack. We discussed code injection attack in our problem description chapter.

\[
\text{SELECT forename, tel, fax, email FROM personal WHERE surname = 'Sharp';}
\]

The above written SQL statement receives input from user in the form of surname which is used to construct query to retrieve forename, telephone, fax and email of user with given surname. When this SQL statement is parsed the XML document which is obtained is given in figure 7.6
7.3.1 Generating XML representing SQL at runtime

At runtime if attacker enters malicious code i.e., instead of entering surname the attacker enters another SQL statement with password field instead of forename in Select statement (based on our second problem description) resulting SQL statement which would be sent to database server for execution would be:

SELECT forenames,tel,fax,email FROM personal
WHERE surname=`'
SELECT password,tel,fax,email FROM personal
WHERE surname=`Sharp';
When this statement is parsed by our parser the XML which is being generated is given in figure 7.7. Here there two SQL statement in one query request. The parser which we wrote is able to identify this type of query and groups them under root node INPUT.

```xml
<INPUT>
  <SQL>
    <SELECT>
      <ID>forename</IDENTIFIER>
      <ID>tel</ID>
      <ID>fax</ID>
      <ID>email</ID>
    </FROM>
    <WHERE>
      <ID> surname </ID>
      <COMP> = </COMP>
      <ID> Sharp </ID>
    </WHERE>
    </SELECT>
  </SQL>
  <SQL>
    <SELECT>
      <ID>password</ID>
      <ID>tel</ID>
      <ID>fax</ID>
      <ID>email</ID>
    </FROM>
    <WHERE>
      <ID> surname </ID>
      <COMP> = </COMP>
      <ID> Sharp </ID>
    </WHERE>
    </SELECT>
  </SQL>
</INPUT>
```

Figure 7.7 XML representation of malicious statement containing multiple SQL statement
Static and runtime model are converted from their XML format to DOM tree format for comparison, the DOMs which are obtained are compared to check if there is any change in structure between static and runtime model.

### 7.3.2 Comparing XML structure for static and runtime SQL for preventing code injection attack

![Figure 7.8 DOM representation of XML given in figure 7.6](image)

Figure 7.8 DOM representation of XML given in figure 7.6
As seen from figure 7.8 there is single SQL statement in query, grouped under root node INPUT. DOM representation of XML representing runtime statement is given in figure 7.9.

Figure 7.9 DOM representation of XML given in figure 5.7

It is clear from figure 7.8 and figure 7.9 that root node of static XML DOM has one child node "SQL" where as root node of dynamic XML DOM has two child node. This difference is detected by matching algorithm when it detects that root
node of both DOM tree has different number of children. This prevents malicious code from reaching database server and hence the attack is thwarted.

### 7.4 Performance Evaluation

Performance was evaluated on AMD Athlon processor (1.6 GHz) with 512MB RAM running Windows XP. We created sample database in Microsoft Sql Server 2000. The J2EE application was run in Tomcat server 5.5.9 which comes bundled with NetBeans software. To match real world scenario all JSP were compiled into servlets. The web server, database server and client were in same local machine. We used IBM page detailer software to determine response time of application. We computed response time for five different queries once for web application with our security architecture embedded in it and once without our security architecture. A graph showing comparison between web application secured with our architecture and web application without our architecture was plotted. The graph is given in figure 7.10.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Maximum</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain SQL</td>
<td>852.5</td>
<td>935.7</td>
<td>903.2</td>
<td>908.9</td>
</tr>
<tr>
<td>Secured SQL</td>
<td>949.70</td>
<td>1118.0</td>
<td>1001.00</td>
<td>986.80</td>
</tr>
</tbody>
</table>

**Table 7.0 Response time (in millisecond)**
Table 5.0 shows statistical figure for response time for different sample for both kind of system.

![Figure 7.10 Response time of plain SQL versus Secured SQL](image)

As seen from graph, our system adds 10-12 percent overhead on existing system. The test was performed on local machine, in real world the typical response for most application lies between few seconds to tens of seconds depending upon the type of application, the small overhead that our system adds to existing web application won't be noticeable. The overhead results due to parsing of SQL statement at runtime and comparison of static and dynamic model of SQL statements represented using XML. Computation of static model does not add any overhead. To improve efficiency, the comparison algorithm returns false the moment it finds
difference in structure of XML document. In that it does not traverse rest of the DOM tree.
CHAPTER 8

Conclusions

In this thesis we presented different types of SQL injection attacks and developed an architecture to prevent it. SQL injection attacks are frequently carried out by hacker to retrieve unauthorized information. These attacks typically alter the syntactic structure of query. Based on this fact we developed a security system which uses static and runtime analysis of SQL statement to prevent SQL injection attacks. We tested our security system for different cases and it worked correctly to prevent SQL injection attack. Our approach allows us to use it on existing web application without much modification in existing code. Performance overhead of our system is less than 12 percent.

Future work

We implemented our algorithm in java and tested it for small J2EE application. Further testing needs to be done on online web application to check its performance and scalability. Currently our system works for J2EE based web application; further works needs to be done to make it a generalized system which can work for .NET and PHP based applications. Another future work would be to implement same technique for XPATH injection attacks. XPath is used to query XML database. XML database is represented using an XML document and XPath query is similar to an SQL query. XPath queries are used for search requests, for
login processing, for data retrieval, and other lightweight database tasks. XPath injection takes place in web site, which constructs XPath from User input.
REFERENCES


Chiba, Japan, June 2005.


