DEPLOYMENT OF REVERSE PROXY FOR THE MITIGATION OF SQL INJECTION ATTACKS USING INPUT-DATA CLEANSING ALGORITHM

S. Fouzul Hidhaya[1, 2] and Angelina Geetha[1, 3]

[1] Department of Computer science and Engineering,
B.S. Abdur Rahman University,
Chennai, Tamilnadu, India.
[2] fouzul_hameed@yahoo.com
[3] anggeetha@yahoo.com

ABSTRACT

Internet has eased the life of human in numerous ways, but the drawbacks like the intrusions that are attached with the Internet applications sustains the growth of these applications. Hackers find new methods to intrude the applications and the web application vulnerability reported is increasing year after year. One such major vulnerability is the SQL Injection attacks (SQLIA). Since SQLIA contributes 25% of the total Internet attacks, much research is being carried out in this area. In this paper we propose a method to detect the SQL injection. We deploy a Reverse proxy that uses the input-data cleansing algorithm to mitigate SQL Injection Attack. This system has been tested on standard test bed applications and our work has shown significant improvement in detecting and curbing the SQLIA.

KEYWORDS

SQL Injection, SQL attack, Security threats, Web application vulnerability.

1. INTRODUCTION

The glory of Internet and its merits are being highly masked by the drawback associated with it. Of them the prime issue is Internet vulnerability, leading to data modification and data thefts. Many Web applications store the data in the data base and retrieve and update information as needed. These applications are highly vulnerable to many types of attacks, one of them being SQL injection Attacks (SQLIA).

The servers that host the web applications retrieve the information from the database and respond to user's request. This poses a high risk problem to the web applications and the data available. To avoid the potential manipulation of the data by the end user, an internal network is formed by coupling the server with reverse proxy.

A reverse proxy is a type of proxy server that retrieves information from the server. The existence of a proxy server may not be known to the end user. The job of checking the request for SQL injection attack could be offloaded from the server to the reverse proxy.

A SQL injection attack occurs when an attacker causes the web application to generate SQL queries that are functionally different from what the user interface programmer intended. For example, consider an application dealing with author details.

A typical SQL statement looks like this:

select id, firstname, lastname from authors;

This statement will retrieve the 'id', 'forename' and 'surname' columns from the 'authors' table, returning all rows in the table. The 'result set' could be restricted to a specific 'author' using ‘where’ clause.

select id, firstname, lastname from authors where firstname = 'James' and lastname = ‘Baker’;

An important point to note here is that the string literals 'James' and 'Baker' are delimited with single quotes. Here the literals are given by the user and so they could be modified. They become the vulnerable area in the application. Now, to drop the table called ‘authors’, a vulnerable literal can be injected into the statement as given below.

Firstname: Jam'; drop table authors--
lastname:

Now the statement becomes,

select id, firstname, lastname from authors where firstname = '*Jam' ; drop table authors***-- and lastname = ' ';**

and this is executed.
Since the first name ends with delimiter ‘ and - - is given at the end of the input, all other command following the - - is neglected. The output of this command is the deletion of the table named ‘authors’, which is not the intended result from a server database.

The objective of this work is to handle SQLIA in any form. An Input-data cleansing algorithm has been designed and implemented in a reverse proxy to effectively curb SQLIA.

2. RELATED WORK

SQL language being a very rich language, paves way for a number of attacks. David Litchfield [1], in his paper classifies the attacks into 3 types: in-band, out-of-band, inference attack. In-band attack refers to extracting of data over the same channel between the server and the client. Out-of-band attack uses different channels to extract data. Inference attack is done by just trying to get error message from the server or invoking the server to return messages that would give out information about the server, the database and the application. In this paper, he discusses how data could be collected using inference.

Allaire [2] in 1999, published a note discussing the dangers of "Multiple SQL statements in dynamic queries". They had for the first time explained how an attack on the database would be possible using appending malicious queries to the existing queries.

The term SQL injection has been believed to be first used in "SQL Injection FAQ" published by Chip Andrews [3]. He, in his FAQ section explains all the facts about the SQL injection.

The black box testing methodology used in WAVES[4], which uses a web crawler to identify all points in web application that can be used to inject SQLIAs. It uses machine learning approaches to guide its testing.

Static code checkers like the JDBC-checker [5] is a technique for statically checking the type correctness of dynamically generated SQL queries. This will be able to detect only one type of SQL vulnerability caused by improper type checking of input.

Combined static and dynamic analysis like the AMNESIA [6] is a model based technique that combines static analysis and runtime monitoring. The key intuition behind the approach is based on two issues. Firstly the source code contains enough information to infer models of the expected, legitimate SQL queries generated by the application. Secondly an SQLIA, by injecting additional SQL statements into a query, would violate such a model. In its static part, this technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, the technique monitors the dynamically generated queries at runtime and checks them for compliance with the statically-generated model.

SQLGuard [7] and SQLCheck [8] also check queries at runtime to see if they conform to a model of expected queries. Taint based approaches like the WebSSARI [9] detects input-validation-related errors using information flow analysis. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. Livshits and Lam [10] use information flow technique to detect when tainted input has been used to construct a SQL query.

Security gateway [11] is a proxy filtering system that enforces input validation rules on the data flowing to a web application. SQLRand [12] is an approach based on instruction-set randomization. It allows developers to create queries using randomized instruction instead of normal SQL keywords. A proxy-filter intercepts queries to the database and de-randomizes the keywords.

William G.J. Halfond, Jeremy Viegas, and Alessandro Orso [13] presented an extensive review of the different types of SQL injection attacks known to date. For each type of attack, they provide descriptions and examples of how attacks of that type could be performed. They also present and analyze existing detection and prevention techniques against SQL injection attacks.

Ofer Maor and Amichai Shulman [14] in their paper give a detailed report of all the techniques used to evade SQL Injection signatures. Signatures are standard forms in which an SQL manipulation could be done to dynamically change the meaning of the query in the application. They give an overview of all the signatures used to protect the server from SQLIA and their counter techniques used to evade those techniques.

Yonghee Shin and Laurie Williams [15] give a survey study of all the papers that deal with SQLIA and Cross-site scripting attack. They categorize the detection methods and evaluation criteria of the techniques. They have surveys around 21 papers and compares among the techniques used to detect cross-site scripting and SQLIA. Stephen Kost [16] in his paper has categorized SQLIA into 4 main categories namely SQL Manipulation, Code injection, Function call Injection and Buffer Overflows. This paper deals with curbing, SQL manipulation and code injection attacks.

Konstantinos Kemalis and Theodoros Tzouramanis [17] developed a prototype SQL injection detection system (SQLIDS). The system monitors Java-based applications and detects SQL injection attacks in real time. The detection technique is based on the assumption that injected SQL commands have differences in their structure with regard to the expected SQL commands that are built by the scripts of the web application.

Ben Smith Et. al [18] in their work examines two input validation vulnerabilities, SQL injection vulnerability and error message vulnerability in four open source applications. They assessed the effectiveness of system and unit level testing of web applications to reveal both the type of vulnerabilities when used with iterative test automation.

In this paper we have designed an input-data cleansing algorithm that uses MD5 hashing to curb SQLIA attempted through the entry form and uses regular expression to curb SQLIA attempted through the URL. The System uses a sanitizing application in proxy server that will sanitize the request before it is being forwarded to the main server and the database.

3. SYSTEM ARCHITECTURE

The architecture of the system is illustrated in Figure 1. In a client server model, a reverse proxy server is placed, in between the client and the server. The presence of the proxy server is not known to the user. The sanitizing application is placed in the Reverse proxy server.

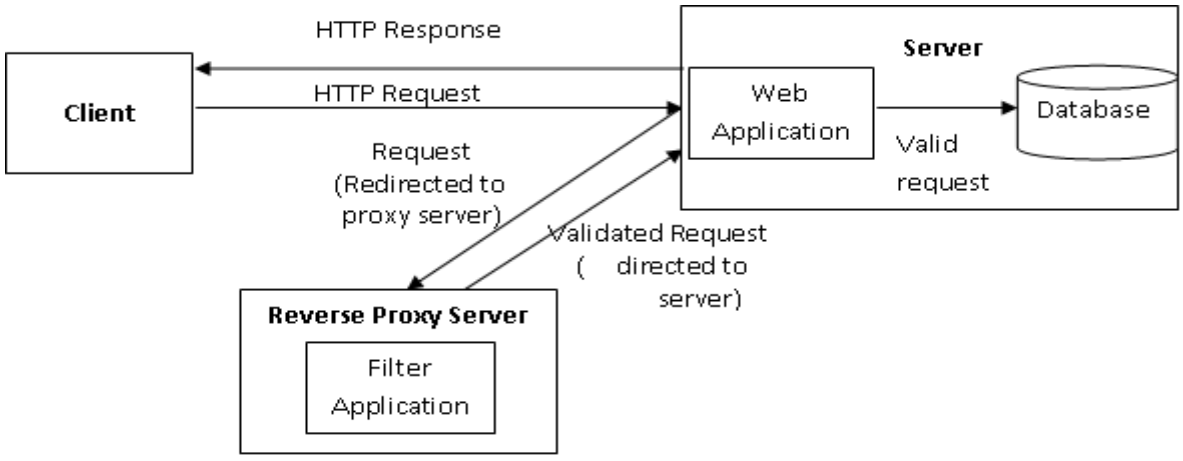

Figure 1. System Architecture

A reverse proxy is used to sanitize the request from the user. When the request becomes high, more reverse proxy's can be used to handle the request. This enables the system to maintain a low response time, even at high load.

The general work of the system is as follows:

1. The client sends the request to the server.
2. The request is redirected to the reverse proxy.
3. The sanitizing application in the proxy server extracts the URL from the HTTP and the user data from the SQL statement.
    a. The URL is send to the signature check
    b. The user data (Using prototype query model) is encrypted using the MD5 hash.
4. The sanitizing application sends the validated URL and hashed user data to the web application in the server.
5. The filter in the server denies the request if the sanitizing application had marked the URL request malicious.
6. If the URL is found to be benign, then the hashed value is send to the database of the web application.
7. If the hashed user data matches the stored hash value in the database, then the data is retrieved and the user gains access to the account.
8. Else the user is denied access. Figure 2 gives the flowchart of the system.

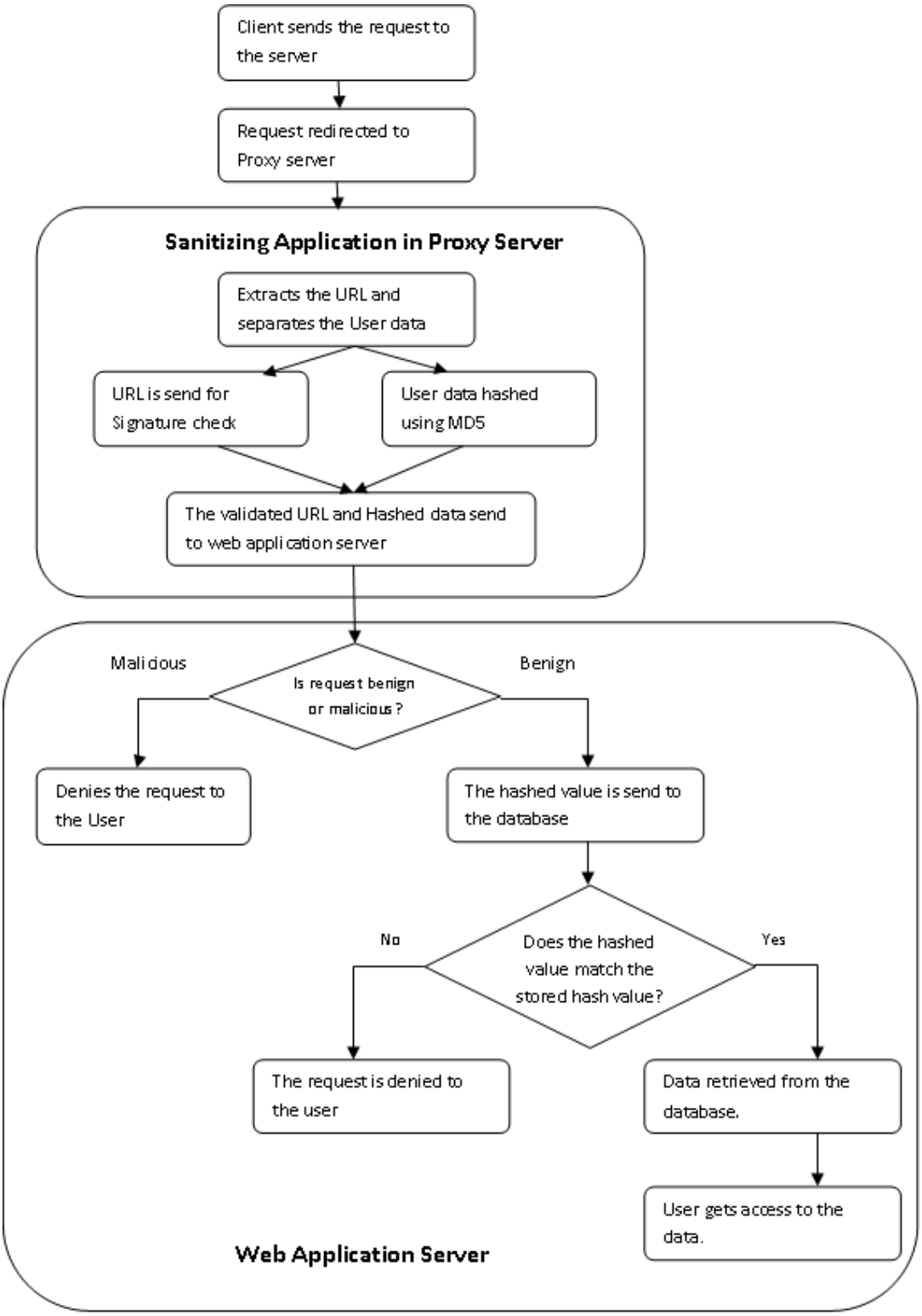

Figure 2. Flowchart of the System

4. THE SANITIZING APPLICATION

The sanitizing application uses the input-data cleansing algorithm to sanitize the user input.

4.1. Input-Data Cleansing Algorithm (IDC algorithm):

Step 1:

Extract the SQL statement from the HTTP;
Parse the SQL statement into Tokens-*query*;
While (not empty of *query*)
  Convert into XML format using XML Schema;
  Add to list - *XMLquery*;
For (every data in prototype document)
  Check if (*XMLquery* = prototype model in document)
    Extract the user input data;

Step 2:

Parse the user data into Tokens-*tok*;
While (not empty of *tok*)
  Check if *tok* ≠ reserved SQL Keyword
    Move *tok* to User data Array-*UDA*;

Step 3:

For (every data in *UDA*)
  Convert to Corresponding MD5 and store in *MD5-UDA*.

Step 4:

Extract the URL from HTTP;
Parse the URL into Tokens-*toks;*
While (not empty of *toks*)
  Check if (URL = Benign using the signature check)
    Set the *flag* to *continue*;
  Else
    Set the *flag* to *deny*;

Step 5:

Send the *MD5-UDA* and *flag* to Web application Server;

4.2. Input-Data Cleansing Algorithm Details

4.2.1. Extracting user data from SQL statement:

The SQL statement is extracted from the HTTP request and the query is tokenized. The tokenized query is then compared with the prototype document. A prototype document consists of all the SQL queries from the Web application. The query tokens are transformed into XML format. The XSL's pattern matching algorithm is used to find the prototype model corresponding to the received Query. This method has been adapted in the previous work COMPVAL [19].
XSL's Pattern Matching: The query is first analyzed and tokenized as elements. The prototype document contains the query pertained to that particular application. For example the input query is,

SELECT * FROM members WHERE login='admin' AND password='XYZ' OR '1=1'
When this query is received this is converted into XML format using a XML schema. The resulting XML would be,

```
<SELECT>
  <*>
   <FROM>
    <members>
      <WHERE login= 'admin'>
        <AND password= XYZ>
          <OR 1=1>
           </OR>
         </AND>
       </WHERE>
      </members>
    </FROM>
  </*>
</SELECT>
```

Using the pattern matching the elements is searched such that the nested elements are similar to query tokens. The corresponding matching XML mapping is,

```
<SELECT>
  <identifier>
   <FROM>
    <identifier>
      <WHERE id_list= 'userip'>
        <AND id_list='userip'>
        </AND>
       </WHERE>
      </identifier>
    </FROM>
  </identifier>
</SELECT>
```

When the match is found, the corresponding prototype query would be,

SELECT identifier FROM identifier WHERE identifier op 'userip' AND identifier op 'userip'

which will be used to identify the user input data . The extra XML tags other than those in the prototype will be considered as user input. This search is less time consuming because the search is based on text and string comparison. The time complexity is O(n). This helps in increasing the effectiveness of the program and reduces the latency time.

4.2.2. Encrypting the user data into MD5 hash

The user data extracted from the extraction phase is then encrypted using the MD5 hash function.

4.2.3. Signature check using regular expressions

All the possible forms of SQL injection manipulation are stored in the signature check in the form of regular expressions. The URL is extracted from the HTTP request and the URL is tokenized. These tokens are checked using the regular expressions. If they contain any form of the signature that has been defined as SQL injection then the request is marked as malicious else it is marked as benign.

5. IMPLEMENTATION

This system implements the IDC algorithm in the automated sanitizing application using Java. We have used 4 systems in the lab setup connected through LAN. One system is considered as the web application server. We set up two systems for the proxy server which has the automated sanitizing application installed. One system acts as the client. On the server an Eclipse integrated development Environment (IDE) runs the open source project. On the server gateway a filter program is installed. This filter application redirects the request from the user to the proxy server. For each request the server chooses one of the two proxy server alternatively. This is done to minimize the loading on a particular proxy server which might slow down the process.

In each of the proxy server the sanitizing application is executed in the Eclipse IDE. When the redirected request from the server reaches the sanitizing application the IDC Algorithm is triggered. As a first step, the SQL query and the URL is extracted from the HTTP. The SQL query is processed using the XSL's pattern matching and the prototype document. The user data is separated from the query. The URL is passed on to the signature check, which uses the regular expression to validate the URL.

The following signature checks are done on the URL's extracted from the HTTP request.

1. Query delimiter (--)
2. White Spaces
3. Comment delimiter (/* */)
4. EXEC keyword
5. UTF coding
6. Scanning for query with signature OR followed same characters before and after '='.
7. Dropping meta characters (and their encoding) like ; ,(,), >, <, %, +,= and @

8. Use of 'IN', 'BETWEEN' after 'OR'.
9. Use of SQL keywords. Just looking into the keywords will bring about a lot of false positives. So the context before and after the keyword is also checked.

The user data is converted into its corresponding hash value using the MD5 algorithm. The hash value and the validated URL are then directed back to the server.

Depending on the validation results the filter on the web application server decides whether to continue with the request or to deny the request. If the URL is benign the URL and the hash value is forwarded to the web application on the server system. The web application sends the hash value to the database and the value are checked. If the values match, then the user gains access. Else the request is denied. The database used with the web application is MySQL.

6. EVALUATION

This system was tested on 4 open source projects. The open source projects that was considered for this study, was taken from *gotocode.com.* The four projects that were taken into study were Online Bookstore, Online portal, Employee directory, registration form. We used Burp suite [20] as an attacking tool. Our system was able to detect all the intrusions injected by burp suite and was able to achieve 100% detection rate. The total number of SQL injections by the Burp suite and the total number of detections by our system defining the detection rate is stated in Table 1. Figure 3 and Figure 4 shows the response of the system when a malicious input is provided in the input form. Figure 5 and Figure 6 shows the response when a malicious URL is given.

Table 1. Detection Rate

Web Application	No. of SQL Injection Attacks	No. of Detections	Detection Rate
Portal	276	276	100%
Employee Directory	238	238	100%
Book store	197	197	100%
Registration Form	419	419	100%

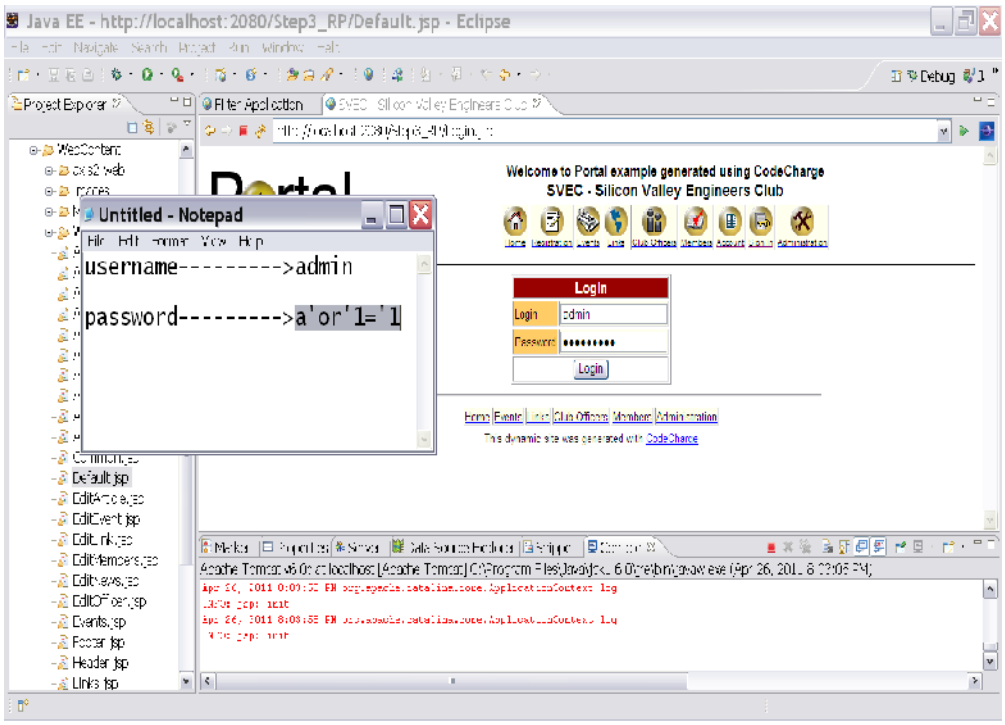

Figure 3. Malicious input provided to the Application.

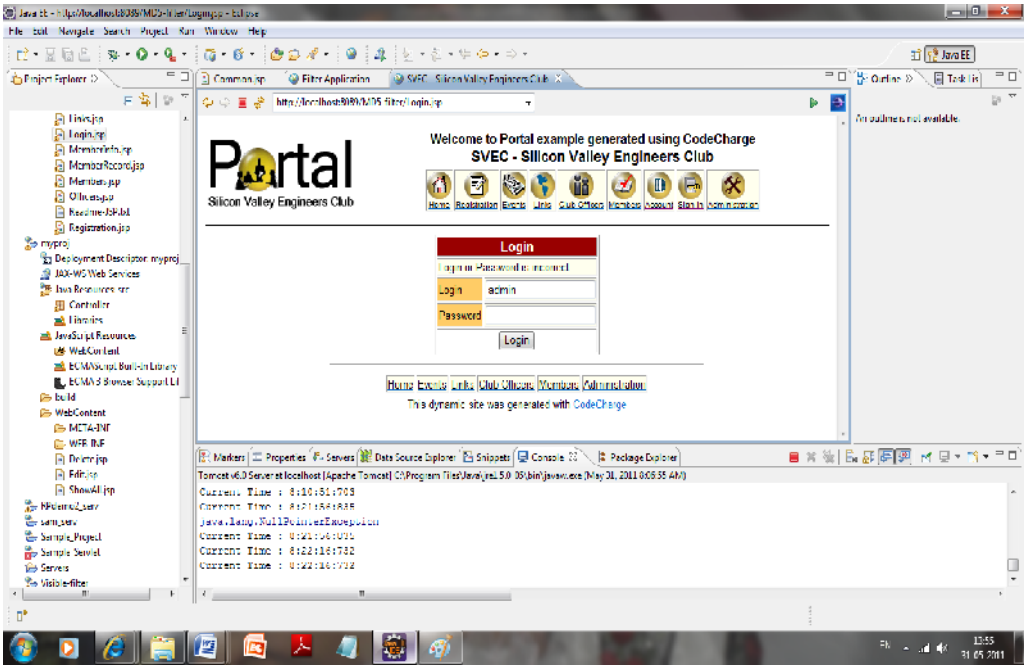

Figure 4. Access denied to the malicious input.

Figure 5. Malicious URL provided to the application.

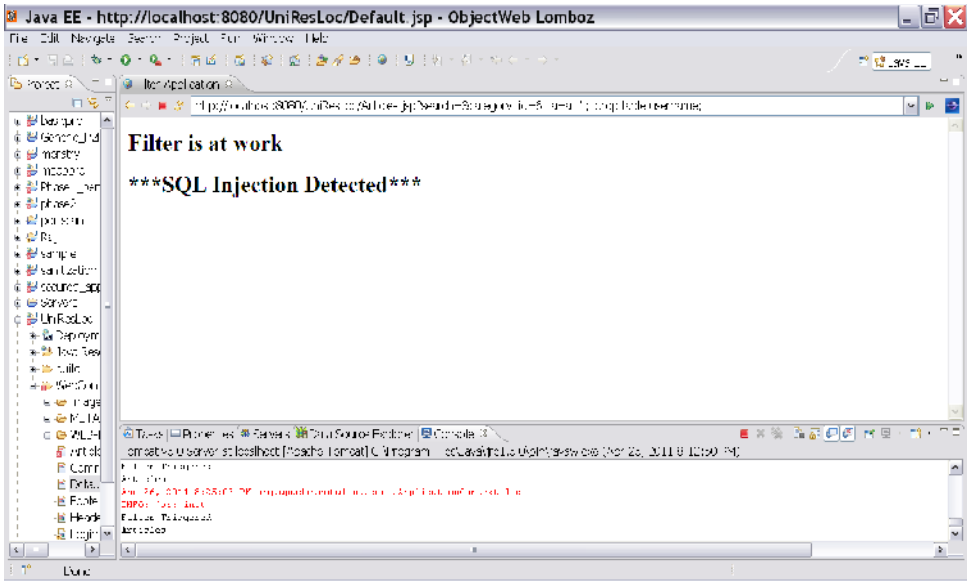

Figure 6. Accesss Denied to the maicious URL

7. ANALYSIS AND RESULT

We have analyzed our system and other methodologies that are used to curb SQLIA. The detailed analysis is shown in Table 2. The system was run under light load condition, medium load condition and heavy load condition. The time taken for the response with our system's Intrusion Prevention proxy (IP proxy) and without the Intrusion Prevention proxy was noted in Nanoseconds. Under Light load condition 5 requests from client system was send to the server. The results are as shown in Figure 7.

Under medium load 50 requests was send from client system using threads. The results are as shown in Figure 8. For heavy load 1000 requests was send using client system. The results are as shown in Figure 9. The time taken did not show much difference for light load and medium load condition. For heavy load condition, there was a slight difference in nanoseconds.

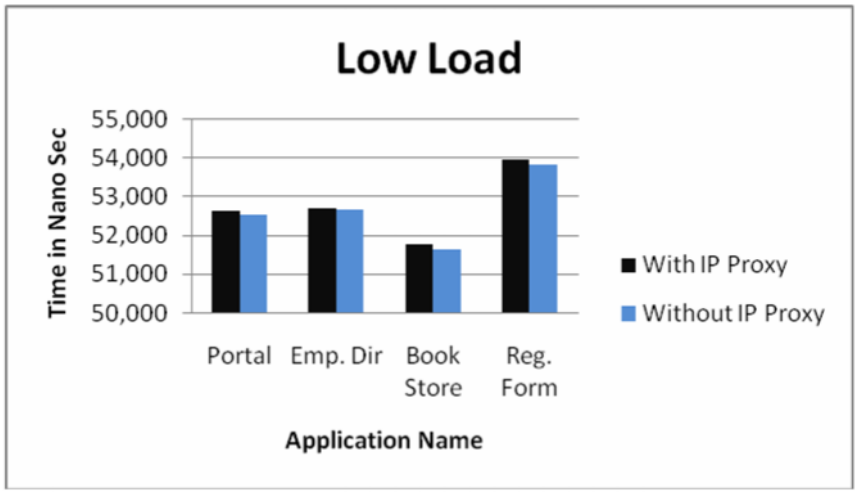

Figure 7. Low load

Table 2. Analysis of methodologies curbing SQLIA

Methodology	**Change in source Code**	**Detection/Mitigation of attack**
WAVES[4]	Not necessary	Automatized/ report generated
JDBC-Checker[5]	Needed for automatic prevention of attack.	Can be automatized.
AMNESIA[6]	Not necessary	Fully automatized
SQLGuard[7]	Necessary	Fully automatized
SQLCheck[8]	Necessary	Partially automatized
WebSSARI[9]	Necessary	Partially Automatized
Livshits and Lam[10]	Not necessary	Manual assistance needed
Security Gateway[11]	Not needed	Manual detection / automatized Mitigation
SQLRand[12]	Necessary	Fully automatized
SQL-IDS[17]	Not necessary	Fully Automatized
Idea[18]	Not necessary	A study to expose vulnerabilities
COMPVAL[19]	Not necessary	Fully automated
Proposed DC algorithm	Not necessary	Fully automated

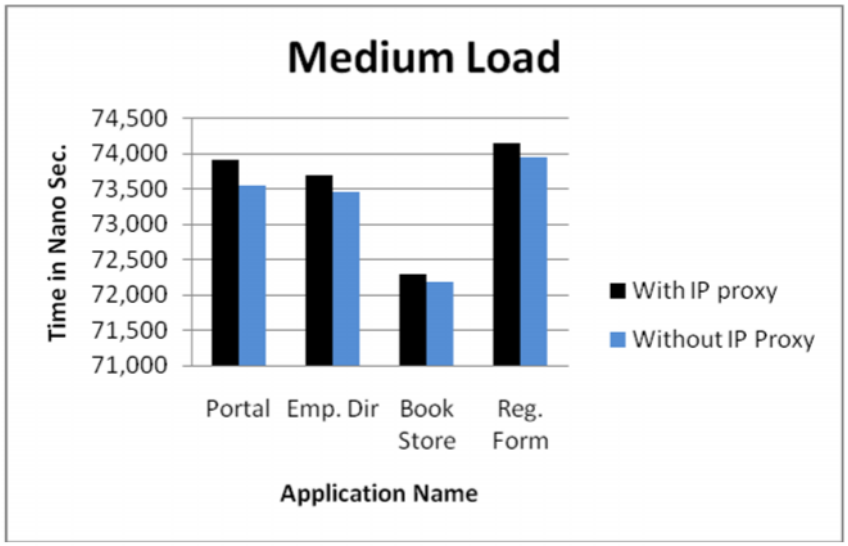

Figure 8. Medium Load

The system using the proxy server protection was responding a little slower than the other system, but had full protection against SQL injection attacks. If we increase the number of proxy server to four then the server was able to handle the request with an increased pace. We have not yet worked on optimization of the system. We believe, after optimization of the system, the performance will improve.

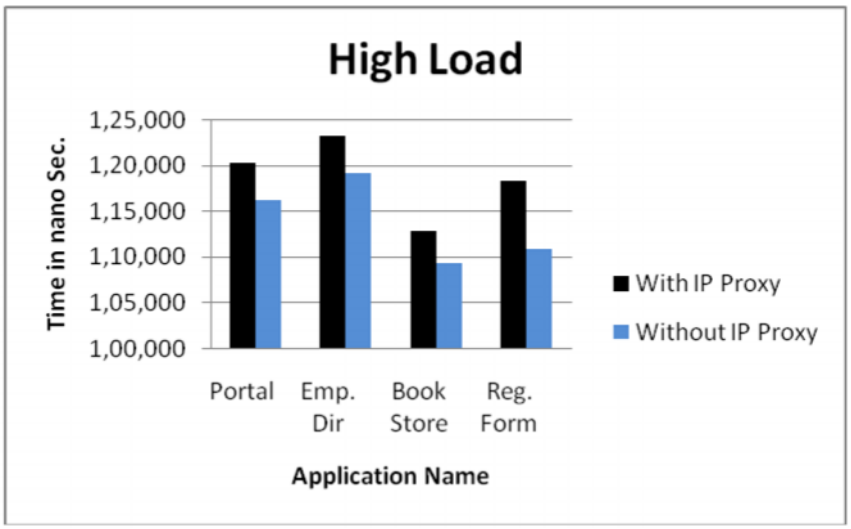

Figure 9. High load

8. CONCLUSION

The novel system with intrusion prevention proxy has proved to be effective in detecting the SQL injection attacks and preventing the attacks from penetrating the web application. This system does not do any changes in the source code of the application. The detection and mitigation of the attack is fully automated. By increasing the number of proxy servers the web application can

handle any number of requests without obvious delay in time and still can protect the application from SQL injection attack. In future work, the focus will be on optimization of the system and removing the vulnerable points in the application itself, in addition to detection and studying alternate techniques for detection and mitigation of SQL injection attacks.

REFERENCES

[1] David Litchfield, (2005) "Data-mining with SQL Injection and Inference", Next Generation Security software Ltd., White Paper.

[2] Allaire Security Bulletin, (1999) "Multiple SQL statements in dynamic queries".

[3] Chip Andrews, "SQL Injection FAQs", http://www.sqlsecurity.com/FAQs/SQLInjectionFAQ/tabid/56/Default.aspx

[4] Y.Huang, F. Huang, T.Lin and C.Tsai, (2003) "Web Application Security Assessment by Fault Injection and Behavior Monitoring", Proc. International World Wide Web Conference '03, pp. 148 - 159.

[5] C.Gould, Z.Su and P.Devanbu, (2004) "JDBC Checker: A Static Analysis Tool for SQL/JDBC Application", Proc. International Conference on Software Engineering '04, pp.697-698.

[6] W. G. Halfond and A. Orso, (2005) "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks", Proc. ACM International Conference on Automated Software Engineering '05, pp. 174-183.

[7] Gregory Buehrer, Bruce W. Weide and Paolo A. G. Sivilotti, (2005) "Using Parse Tree Validation to Prevent SQL Injection Attacks", Proc. International Workshop on Software Engineering and Middleware, pp. 106-113.

[8] Zhendong Su and Gary Wassermann, (2006) "The Essence of Command Injection Attacks in Web Applications", Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages '06, pp.372-382.

[9] Y.Huang, F.Yu, C. Hang,C.H. Tsai, D.T.Lee and S.Y.Kuo, (2004) "Securing Web Application Code by Static Analysis and Runtime Protection", Proc. International World Wide Web Conference '04, pp. 40-52.

[10] V.B. Livshits and M.S. Lam, (2005) "Finding Security Errors in Java Programs with Static Analysis", Proc. Usenix Security Symposium '05, pp. 271-286.

[11] D.Scott and R.Sharps, (2002) "Abstracting Application-level Web Security", Proc. International Conference on the World Wide Web '02, pp. 396-407.

[12] S.W. Boyd and A.D. Keromytis, (2004) "SQLrand: Preventing SQL Injection Attacks", Proc. 2nd Applied Cryptography and Network Security (ACNS) Conference, pp. 292-302.

[13] W. Halfond, J. Vigeas and A.Orso, (2006) "A Classification of SQL Injection Attacks and Counter Measures", Proc. International Symposium on Secure Software Engineering '06.

[14] Ofer Maor and Amichai Shulman, (2003) "SQL injection signature evasion", Imperva Inc., White paper.

[15] Yonghee Shin and Laurie Williams, (2008) "Toward A Taxonomy of Techniques to Detect Cross-site Scripting and SQL Injection Vulnerabilities", NC state Computer science: Technical report.

[16] Stephen Kost, (2004) "An introduction to SQL injection attacks for Oracle developers", Integrity Corporation, White paper.

[17] Konstantinos Kemalis and Theodoros Tzouramanis, (2008) "SQL-IDS: a specification-based approach for SQL-injection detection", Proc. 2008 ACM symposium on Applied computing, pp.2153 - 2158.

[18] Ben Smith, Laurie Williams and Andrew Austin, (2010) "Idea: Using System Level Testing for Revealing SQL Injection-Related Error Message Information Leaks", Engineering Secure Software and Systems, Springer, Lecture Notes in Computer Science, Volume 5965/2010, 192-200.

[19] S. Fouzul Hidhaya and Angelina Geetha, (2010) "COMPVAL – A system to mitigate SQLIA", Proc. International Conference on Computer, Communication and Intelligence ICCCI'10, PP.337-342.

[20] Burp suite, http://portswigger.net/burp/