Web Based Intrusion Detection System for SQLIA

Ayesha Siddiqa[1]

[1] *Dept. of Computer Science & Engineering*
JNNCE

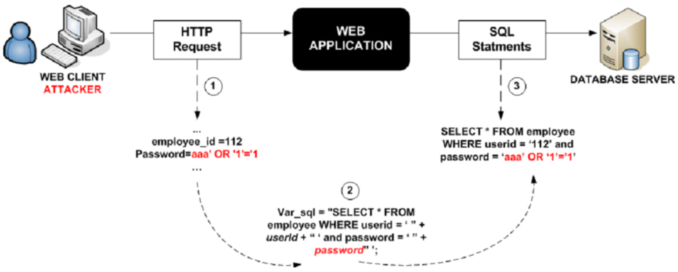
Abstract - SQL Injection Attack (SQLIA) refers to an injection attack wherein an attacker can execute malicious SQL statements that control a web application's database server. By leveraging SQL Injection vulnerability, given the right circumstances, an attacker can use it to bypass a web application's authentication and authorization mechanisms and retrieve the contents of an entire database. SQL Injection can also be used to add, modify and delete records in a database, affecting data integrity. The main idea of our work is to allow developers the freedom to write and execute code without having to worry about these attacks. In this paper we propose a Web Based Intrusion Detection System for SQLIA to extract a SQL query connecting to database from a PHP file. The structure of the query under observation will be converted to XML file and compared against the legitimate queries stored in the XML file using association rule mining thus minimizing attacks. WEBIDS is expected to reduce the time and manual effort as it only focuses on fragments that are vulnerable for attacks.

Key Words: XML Rule Mining, PHP, SQL injection,

1.INTRODUCTION

SQL injection is the command consisting of some SQLs (SQL statements) that are used to control information within a database. Such website which referring certain database(s) applies this mechanism: in such a website, web applications will return the user the results dynamically reflected when the applications request to the database according to a user's inputs. What if the web application has vulnerability relevant to SQL injection, an adversary may inject malicious SQL statements so that the information within that database may be manipulated fraudulently This malicious accessing method is specifically referred as SQL injection attack.

Database-driven web applications have become widely deployed on the Internet, and organizations use them to provide a broad range of services to their customers. These applications, and their underlying databases, often contain confidential, or even sensitive, information, such as customer and financial records. However, as the availability of these applications has increased, there has been a corresponding increase in the number and sophistication of attacks that target them. One of the most serious types of attack against web applications is SQL injection. In fact, the Open Web Application Security Project (OWASP), an international organization of web developers, has placed SQL injection attack (SQLIA) at the top of the top ten vulnerabilities that a web application can have [1]. Similarly, software companies such as Microsoft have cited SQLIAs as one of the most critical vulnerabilities that software developers must address [2]. As the name implies, this type of attack is directed toward database layer of the web applications. Mechanism of SQL injection is illustrated in Fig.1 [3]


WEB CLIENT
ATTACKER
HTTP Request
WEB APPLICATION
SQL Statments
DATABASE SERVER
1
...
employee_id =112
Password=aaa' OR '1'='1
...
2
Var_sql = "SELECT * FROM employee WHERE userid = ' " + userid + " ' and password = ' " + password" ';
3
SELECT * FROM employee WHERE userid = '112' and password = 'aaa' OR '1'='1'

In this paper, we propose a System that combines the two IDS techniques, Embezzle and Eccentricity detection techniques, to defend against SQLIA. The main idea of Web Based Intrusion Detection system (WEBIDS) framework is to create a profile for web application that can represent the normal behavior of application users in terms of SQL queries they submit to the database. Database logs can be used to collect these legitimate queries provided that these logs are free of intrusions. We then use an Eccentricity detection model based on data mining techniques to detect queries that deviates from the profile of normal behavior. The queries retrieved from database log are stored in XML file with predefined structure. We choose XML format because it is more structured than flat files, more flexible than matrices, simpler and consume less storage than databases.

Association rules will be applied to this XML file to retrieve relation between each table in the query with each condition in the selection part. These rules represent the profile of normal behavior and any deviation from this profile will be considered attack. In order to better detect SQLIA and to minimize false positive alerts, WEBIDS system as a second step uses misuse technique to detect any change in the structure of the query. Malicious users sometimes don't change the selection clause but add another SQL statement or add specific keywords to the initial query to check the vulnerability of the site to SQLIA or to perform inference attack. Such types of attack are detected in the second step of the detection process. By comparing the structure of the query under test with the corresponding queries in the XML file the previous malicious actions will be detected.

This paper is organized as follows. Section II, includes some related work in web application Vulnerability domains. Section III, provides a detailed description about the System and its components. In Section IV, Results and Discussions. Section V concludes the paper and outlines future work.

II. Literature Survey

Different researches and approaches have been presented to address the problem of web attacks against databases. Considering SQLIA as top most dangerous attacks, as stated in section I, there has been intense research in detection and prevention mechanisms against this attack [4, 5].

A general framework for detecting malicious database transaction patterns using data mining was proposed by Bertino et al. in [6, 7] to mine database logs to form user profiles that can model normal behaviors and identify anomalous transactions in databases with role based access control mechanisms. The system is able to identify intruders by detecting behaviors that differ from the normal behavior of a role in a database. Kamra et al. in [8] illustrated an enhanced model that can also identify intruders in databases where there are no roles associated with each user. It employs clustering techniques to form concise profiles representing normal user behaviors for identifying suspicious database activities. Another approach that checks for the structure of the query to detect malicious database behavior is the work of Bertino et al. in [9]. They proposed a framework based on anomaly detection technique and association rule mining to identify the query that deviates from normal database application behavior.

The problem with this framework is that it produces a lot of rules and represents the queries in very huge matrices which may affect tremendously on the performance of rule extraction. Misuse detection technique have been used by Bandhakavi et al. in [10] to detect SQLIA by discovering the intent of a query dynamically and then comparing the structure of the identified query with normal queries based on the user input with the discovered intent. The problem with this approach is that it has to access the source code of the application and make some modifications to the java virtual machine.

Recently, applicability and scalability of model checking approaches in the domain of web applications started being explored by the research community. In particular, in 2008, a work describing the QED system was published [11]. QED identifies XSS and SQL injection vulnerabilities that arise as a result of the interaction of multiple modules of a servlet-based web application. The system uses explicit model checking to find XSS and SQL injection vulnerabilities and uses a number of heuristics to scale the approach to large applications. To find vulnerability, the tool needs to be supplied with a specification of a vulnerability (written in SQL) and a set of inputs to the application under test. Then, the Java Pathfinder [12] model checker is used to execute the application using a sequence of user requests that are generated based on user input values provided by an analyst. Vulnerability is found when a match to a SQL query is found by the model checker. In general, the approach proposed in this work can be applied to detect vulnerabilities other than taint-based ones if an analyst is able to provide the tool with a specification of a vulnerability specifying patterns of events (such as program method calls) that need to occur on a program path.

Hal fond et al. in [13] developed a technique that uses a model-based approach to detect illegal queries before they are executed on the database. In its static part, the 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 uses runtime monitoring to inspect the dynamically-generated queries and check them against the statically-built model. The system WASP proposed by William et al. in [14] tries to prevent SQL Injection Attack by a method called positive tainting. In positive tainting, the trusted part of the query (static string) is not considered for execution and masked as tainted, while all other inputs are considered. The difficulty in this case is the propagation of taints in a query across function calls especially for the user defined functions which call some other external functions leading to the execution of a tainted query. Different other researches followed the same approach in detection of anomalous SQL query structure in [15, 16].

The contribution of this paper is a System that combines Eccentricity and Embezzle detection technique in order to better detect SQLIA. This System uses association rules with Eccentricity technique to build the normal behaviour of application users and detecting anomalous queries. Moreover, A technique is used to check the structure of the query to detect any malicious actions that cannot be detected using detection technique

III. Proposed Method

WEB Based IDS framework is a database intrusion detection that aims to detect SQLIA at real-time, before queries execution at the database. WEBIDS is simple and really easy to implement. During this technique all the data validations rules are going to be during a secure place. The data validation rules also will be organized into some XML format and that they are referred to as XML-rules. Whenever server receives any input from client the server can verify the whole XML script supported the verification rules already written within the server. XML-rules are going to be written on an individual basis for every kind of incoming XML scripts and therefore the incoming XML script should succeed the validation method. This method can primarily divide the data validation of a web application from the application development division. The developer at present ought not to worry about the SQL injection attacks and data validity. The validation parts of data are going to be maintained by a separate cluster which can manage the XML-rules. This is often conjointly useful as a result of the traditional web developers are going to be utterly unaware regarding the safety rules of the application. WEB based IDS framework combines the two detection techniques: Abnormality and Invade. Figure 2 explains the essential flow of WEBIDS. In a PHP file the database connecting query is fetched and can submit the query then rather than submitting straightforward data, it'll submit all the data in XML format.XML file of query tokens is compared with XML rules by performing the validation. If the validation is false, then it will flag the injection if true it will execute the query.

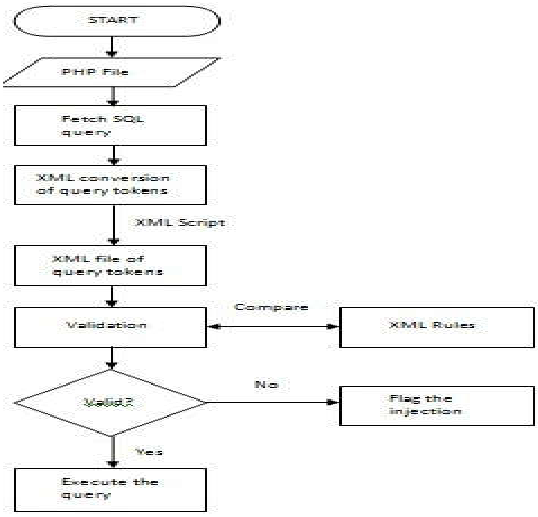


Fig 2.Flow Graph of proposed WEEIDS

The key idea of our system is as follows. We build a repository containing set of legitimate queries submitted from the application user to the database. This repository is a set of training records. We then use an Abnormality detection approach based on data mining technique to build a profile of normal application behavior and indicate queries that deviates from this normal behavior.

In a second step in the detection process, the framework checks for the presence of dangerous keywords in the query if the latter passes the test of Abnormality detection step. We need this step because sometimes the intent of the attacker is to identify the security holes in the site or to infer the structure of the database through the error message returned from the application and this type of SQLIA is called inference [17, 18]. This type of attack cannot be detected through eccentricity technique because it doesn't require change in the conditions of the original query but it will be discovered if the structure of the query is compared against its corresponding query in the repository file.

Based on what previously stated we learn that the System act in two phases: training phase and detection phase. In the training phase the repository file will be created and normal behavior of the application is built. In the detection phase, the framework uses the Eccentricity and Embezzle techniques to discover any SQLIA. In the following subsections we will provide a detailed explanation of the System, its components and how it works.

Training Phase

During the training phase the training records are collected from the queries the application send to the database. The source for obtaining these query traces is the database log provided that the latter is free of intrusions. The training phase flow is illustrated in Fig. 3. The challenge here is that to efficiently encode these queries in order to extract useful features from them and accordingly build the application fingerprint. Unlike approach provided in [19], we choose to encode the queries in XML file. The encoding scheme provided by Bertino et al. in [19] result in a large, dense, sparse matrices which may effect on the mining algorithm. XML is more structured than flat files, is supported by query tools like XQuery and XPath to extract data [20]. It is simpler and consumes less space than relational databases and more flexible than matrices.

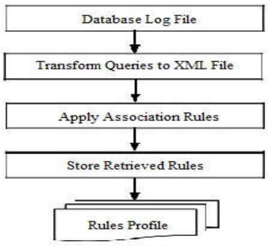


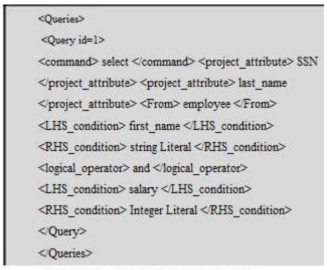
Fig 3 Training Phase Flow

It is important to identify accurately the structure of the XML file that will represent the features extracted from the query that will contribute in building the application fingerprint. Consider the following query:

Select SSN, last_name from employee
where first_name='Suzan' and
salary>5000

The encoding scheme of the previous query in XML file is illustrated in Fig. 4. The main advantage of XML format is that nodes may be duplicated upon need. For example the number of "project attribute" node may differ from one "Query" node to another depending on the query itself. This is why it is more suitable to store queries than databases while maintaining flexibility and simplicity.

The XML file illustrated in Fig. 4 stores the projection attributes, the from clause and the predicate clause in a more detailed way. It is not important to identify the value of the integer or string literal it is important to determine that there is an integer or string literal or there is another attribute in the right hand side. Another file that should be created during the training phase is the signature file that will be used during the misuse detection phase. As stated before this file contains suspicious keywords that may be considered a sign of SQLIA.



```
<Queries>
 <Query id=1>
<command> select </command> <project_attribute> SSN
</project_attribute> <project_attribute> last_name
</project_attribute> <From> employee </From>
<LHS_condition> first_name </LHS_condition>
<RHS_condition> string Literal </RHS_condition>
<logical_operator> and </logical_operator>
<LHS_condition> salary </LHS_condition>
<RHS_condition> Integer Literal </RHS_condition>
</Query>
</Queries>
```

Fig 4 Representation of Query in XML File

The important step in the training phase is to build the profile representing the application normal behavior. We will apply association rules [21] on the XML file to extract rules that represent the normal behavior of application users. The rules extracted represent relationship between each table in the query with each predicate in the selection clause.

This is based on an observation that the static part of the query is the projection attribute and the part that is constructed during execution is the selection part [19]. We here add another item to the static part which are the tables in the from clause. We try to make relation between the static part and the dynamic part and extract rule with support of such relation. Any query that will not match rules extracted and stored in the rules profile will be considered attack. More details about how the rules are extracted are provided in the following subsection.

Abnormality Detection Phase

In the previous subsection, we illustrated how the begin queries are collected and captured in XML file in a form enabling the framework from creating the database behavior profile. We apply association rules on the XML file containing legitimate queries and extract rules that can describe the normal behavior of application users. The idea behind building the profile rule is to apply one of association rules algorithms on previously created XML file to extract relation between each table in the query with each selection attribute excluding the literals.

Thus the rules extracted have the following format:

From → LHS

From → RHS

Example:

Employee → name

Employee → place

The rules that exceed the minimum support will be stored in rules profile. These rules represent the profile of how the application behaves normally. In a typical database application, the input supplied by the user construct the where clause of the query. Meanwhile, the projection clause and the from clause remain static at the run time. So we create a relation between the static and the dynamic part of the query and any change in the where clause by attackers that cannot be derived from the rules profile will be announced as SQLIA. We decided to choose the tables in the from clause from the static part of the query instead of the projection attributes because the former is more general and contain the latter and thus generating less rules and make it easier in comparison. Lets return to our query in the previous subsection and change it a little bit: select SSN, lname from employee where name=' "& fname &" and place= " & emplace". If the attacker needs to retrieve all values from employee table then the following code will be injected to form this new query:

Select SSN, lname from employee where name='' or 1=1

Before executing this query, rules should be extracted first and compared to the rules in the rules profile. The relation between tables and attributes will be compared against rules stored in the profile rules file. The two relations under test from the previous example are:

Employee → name

Employee → 1

The first relation exists in the rules profile but no such rule matches the second relation. So the query is announced as Eccentric query.

Invade Detection Phase

In a second step in the detection process and after the Eccentricity detection phase, comes the role of Embezzle detection. The need to this step comes from the fact that SQLIA doesn't only change the conditions in the query but it also may provide information about the database schema or check the vulnerability of the application to SQL injection. This is done through adding to the query some keywords that may change the behavior of the query or return information about the database through database errors without changing the predicates of the query. In such case, the Eccentricity detection phase will not be able to discover such attack. For example consider the following query:

Select * from employee where SSN=10

If the attacker just adds a single quote at the end of the query, this will result in error message that may inform the attacker that the site is vulnerable to SQLIA. Another example of attack is just adding the keyword "order by" to the query without changing the selection attributes like:

Select * from employee where SSN=10 order by 1

Trying to execute this query several times will give attacker information about the number of attributes in the table. This is why this step is needed in the detection process. Moreover, the framework doesn't announce the query as anomaly just by finding these keywords in the query because it may be part of the legitimate query itself resulting in false positive alarm. This is why the framework checks for the structure of the query under test with the corresponding query stored in XML file. The detection phase flow of the framework in Fig. 5 illustrates this process.

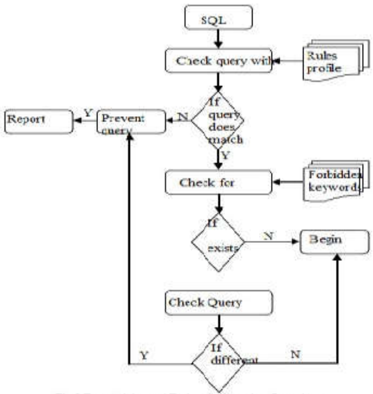


Fig 5 Eccentricity and Embezzle detection flow phase

These suspicious keywords are stored in file called "forbidden keywords". This file contains SQL keywords like single quote, semicolon, union, order by, exec and their hexadecimal representation to avoid the different evasion techniques. After confirming the existence of one or more of these keywords, we use XQuery to retrieve queries from XML file with the same projection attributes and same from clause. Then comparison is done between query under test and the queries retrieved by XQuery from XML file. If there is no match, then the query is announced Eccentric.

IIV. Results and Discussions

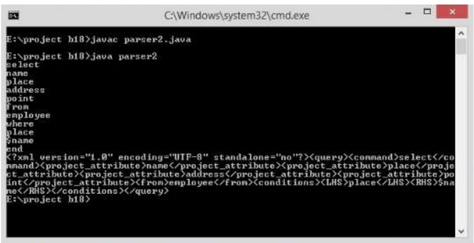
In this section we present results and discussions for eccentricity and embezzle detection. In addition, we provide a working example illustrating how the WEBIDS framework performs the detection.

A. Working Example

We provide in this subsection example of the flow of detection either Eccentricity or Embezzle in this framework. The following represents example of php file submitted from application to the database:

```
<?php
 $name=$_REQUEST['t1'];

mysql_connect("localhost","pw","jnnce");
 mysql_select_db("user");
     $res=mysql_query("select * from contact where name='$name'");
      ................
      ................
      .................
```



```
E:\project b18>javac parser2.java

E:\project b18>java parser2
select
name
place
address
point
from
employee
where
place
$name
end
<?xml version="1.0" encoding="UTF-8" standalone="no"?><query><command>select</co
mmand><project_attribute>name</project_attribute><project_attribute>place</proje
ct_attribute><project_attribute>address</project_attribute><project_attribute>po
int</project_attribute><from>employee</from><conditions><LHS>place</LHS><RHS>$na
me</RHS></conditions></query>
E:\project b18>
```

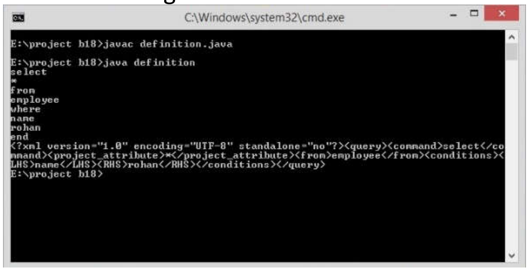
Fig 6 Query is Fetched and converted to XML

In Figure 6 the query is fetched from a php file and Converted to XML

The following represents example of queries submitted from application to database:

- Select * from Employee where name=$name;
- Select * from Employee where place=$place;
- Select * from Employee where salary=$salary;

The representation of the previous queries in XML file is illustrated in Fig. 7.



```
E:\project b18>javac definition.java

E:\project b18>java definition
select
*
from
employee
where
name
rohan
end
<?xml version="1.0" encoding="UTF-8" standalone="no"?><query><command>select</co
mmand><project_attribute>*</project_attribute><from>employee</from><conditions><
LHS>name</LHS><RHS>rohan</RHS></conditions></query>
E:\project b18>
```

Fig 7 Representation of Query

After applying association rule algorithm like for example Priory on this XML file, the resulting rules will be stored in rules profile file in Fig. 7.

Association rules for these definition queries will be as follows:

Employee → name

Employee → place

Employee → salary

These rules will be compared with the actual query association rule. Hence the vulnerability is checked.

Example:

Select * from Employee where name=Mohan;

Employee → name

This query will be ACCEPTED as it matches with the first definition.

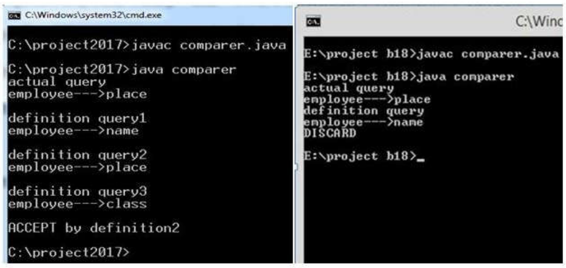
Example:

Select * from Employee where A=A;

Employee → A

This query will be DISCARDED since it is not matched by any definition.

Accept and discard format is demonstrated in Fig 8.



```
C:\project2017>javac comparer.java

C:\project2017>java comparer
actual query
employee--->place

definition query1
employee--->name

definition query2
employee--->place

definition query3
employee--->class

ACCEPT by definition2

C:\project2017>
```

```
E:\project b18>javac comparer.java

E:\project b18>java comparer
actual query
employee--->place
definition query
employee--->name
DISCARD

E:\project b18>_
```

Fig 8 Example of Accept and Discard

In the following we will provide sample of malicious and legitimate queries.

Select product_name, description from product where product_id=5'

The first step in the framework is to identify relation between tables and selection attributes in the query.

Product → product_id

Second, the framework searches in the rules profile for this relation. It already exists. But this is not the end of the detection flow. The second step is to check for suspicious keywords in the query. The query already contains one of the suspicious keyword which is single quote.

So XQuery language is used to extract queries from the XML file with same projection attributes and same from clause. By comparing the structure of the query under test and query returned from the XML file we will find that query contain the single quote and thus it is announced as Eccentric.

V. Conclusion and Future Work

Database intrusion is a major threat to any organization storing valuable and confidential data in databases. We have introduced a System based on Eccentricity and Embezzle detection for discovering SQLIA. Detection is done by validating the SQL queries using general validation procedure based on XML rules and the nature of the injection type. The concepts explained in this work assist the Developer to modify the SQL statements and make the code attack free. We conclude by highlighting the robust features of the efficient WEBIDS, which can detect the error during the development statically and can protect web applications from the future SQL injection.

We believe that the ideas presented in this research work can be further extended to include new injection types to include detection against other attacks like cross site scripting. This work also paves way for the development of vulnerability detection services, which can be used by developers to detect vulnerability spots in the source code. We feel the area of SQL injection vulnerabilities is wide open for research.

References

[1] http://www.owasp.org/index.php, OWASP Top 10-2010 document

[2] M. Howard and D. LeBlanc, “Writing Secure Code”, Microsoft Press,2002

[3] http://www.ipa.go.jp/security/english/virus/press/200805/E_PR200805.html

[4] Kindy, D.A.; Pathan, A.K, “A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques”, in proceedings of IEEE 15th International Symposium on Consumer Electronics (ISCE), 2011

[5] N. Khochare, S. Chalurkar ,S. Kakade, B.B. Meshramm, “Survey on SQL Injection attacks and their countermeasures”, International Journal of Computational Engineering & Management (IJCEM), Vol. 14, October 2011

[6] Bertino, E., Kamra, A, Terzi, E., and Vakali, A, “Intrusion detection in RBAC-administered databases”, in the Proceedings of the 21st Annual Computer Security Applications Conference, 2005.

[7] Kamra A, Bertino, E., and Lebanon, G.,”Mechanisms for Database Intrusion Detection and Response”, in the Proceedings of the 2nd SIGMOD PhD Workshop on Innovative Database Research, 2008

[8] Kamra A, Terzi E., and Bertino, E.,“Detecting anomalous access patterns in relational databases”, the VLDB Journall VoU7, No. 5, pp.1063-1077, 2009

[9] Bertino, E., Kamra, A, and Early, J., “Profiling Database Application to Detect SQL Injection Attacks”, In the Proceedings of 2007 IEEE International Performance, Computing, and Communications Conference, 2007.

[10] Bandhakavi, S., Bisht, P., Madhusudan, P., and Venkatakrishnan V.,

“CANDID: Preventing sql injection attacks using dynamic candidate evaluations”, in the Proceedings of the 14th ACM Conference on Computer and Communications Security, 2007

[11] M. Martin and M. Lam. Automatic Generation of XSS and SQL Injection At-tacks with Goal-Directed Model Checking. In Proceeding of the 17th USENIX Security Symposium, pages 31–43, July 2008

[12] Java pathfinder. http://javapathfinder.sourceforge.net/

[13] Halfond, W. G. and Orso, A , “AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks”, in Proceedings of the 20th

IEEE/ACM international Conference on Automated Software Engineering, 2005

[14] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios, “WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation”, IEEE Transactions on Software Engineering, Vol. 34, No. 1, pp 65-81, 2008

[15] Buehrer, G., Weide, B. w., and Sivilotti, P. A, “Using Parse Tree Validation to Prevent SQL Injection Attacks”, in Proceedings of the 5th international Workshop on Software Engineering and Middleware, 2005

[16] Liu, A, Yuan, Y., Wijesekera, D., and Stavrou, A, “SQLProb:A Proxy-based Architecture towards Preventing SQL Injection Attacks”, in Proceedings of the 2009 ACM Symposium on Applied Computing, 2009

[17] W.G.Halfond, J.Viegas, and A.Orso, “A classification of SQL-Injection Attacks and Countermeasures”, in proceeding of the International

Symposium on Secure Software Engineering (ISSSE), 2006

[18] David Litchfield, “Data-mining with SQL Injection and Inference”,An NGSSoftware Insight Security Research, September 2005

[19] Bertino, E., Kamra, A, and Early, J., “Profiling Database Application to Detect SQL Injection Attacks”, In the Proceedings of 2007 IEEE International Performance, Computing, and Communications Conference, 2007

[20] World Wide Web Consortium. XQuery 1.0: An XML Query Language (W3C Working Draft). http://www.w3.org/TR/2002/WDxquery-20020816, Aug. 2002.

[21] Han J., Kamber M., “Data Mining: Concepts and Techniques”, Maurgan Kaufmann,2ndedition,2000