Runtime Monitoring Framework for SQL Injection Attacks

Ramya Dharam and Sajjan G. Shiva

Abstract—**The increasing use of web applications to provide reliable online services, such as banking, shopping, etc., and to store sensitive user data has made them vulnerable to attacks that target them. In particular, SQL injection, which allows attackers to gain unauthorized access to the database by injecting specially crafted input strings, is one of the most serious threats to web applications. Although researchers and practitioners have proposed various methods to address the SQL injection problem, organizations continue to be its victim, as attackers are successfully able to circumvent the employed techniques. In this paper, we present and evaluate Runtime Monitoring Framework to detect and prevent SQL Injection Attacks on web applications. At its core, the framework leverages the knowledge gained from pre-deployment testing of web applications to identify valid/legal execution paths. Monitors are then developed and instrumented to observe the application's behavior and check it for compliance with the valid/legal execution paths obtained; any deviation in the application's behavior is identified as a possible SQL Injection Attack. We conducted an extensive evaluation of the framework by targeting subject applications with a large number of both legitimate and malicious inputs, and assessed its ability to detect and prevent SQL Injection Attacks. The framework successfully allowed all the legitimate inputs to access the database without generating any false positives, and was able to effectively detect and prevent attacks without generating any false negatives. Moreover, the framework imposed a low runtime overhead on the subject applications compared to other techniques.**

Index Terms—**Basis path testing, data flow testing, runtime monitoring, SQL injection attacks (SQLIAs)**

I. Introduction

Web applications have become popular means of modern information retrieval and interaction, and store sensitive user data such as financial, medical, personal information records, etc., in the back-end database. The increasing use of such applications has made them vulnerable to attacks such as SQL Injection Attacks (SQLIAs), Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Path Traversal Attacks, etc. In June 2013, the Open Web Application Security Project (OWASP) officially released the Top 10 attacks [1] performed on web applications. SQLIAs have been ranked again as the most widely performed attack and a major security threat to web applications. There are many examples of SQLIAs performed on organizations such as Sony [2], LinkedIn [3], Nvidia [4], and Gamigo [5] during recent years causing serious consequences. SQLIAs give attackers direct access to the database underlying an application and allow them to leak sensitive information. Hence, there is an emerging need to protect web applications from such attacks and to assure the confidentiality of user data.

Manuscript received January 4, 2014; revised March 10, 2014.

The authors are with the Department of Computer Science, University of Memphis, Memphis, TN 38152 USA (e-mail: rdharam@memphis.edu, sshiva@memphis.edu).

Web applications are structured as a three-tiered architecture, as shown below in Fig. 1, which consists of a web browser, an application server, and a back-end database server. Such an application will accept input from external users via forms, dynamically construct the database queries using the inputs, dispatch them to the underlying database for execution, and finally retrieve and present the data to the user.

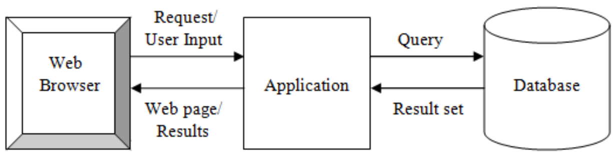


Fig. 1. Web application structure.

SQLIAs, a class of code injection attacks, are performed to gain unauthorized access to sensitive user data residing in the database. They occur when the input, provided by a malicious user, consisting of SQL keywords or operators is not properly validated and is included directly as part of the query. This causes the web application to generate and send a query that in turn results in unintended behavior of the web application, thus causing the loss of confidential user data. For example, consider an Employee Directory application which accepts username and password input strings from users and displays the specific employee details. Such an application will have a back-end database that stores the usernames and passwords of different users. It may contain code such as the following to access and retrieve the data from the database:

*Query = "SELECT * FROM employeeinfo WHERE name = ' "+ request.getParameter ("name") + " ' AND password = ' "+ request.getParameter ("password") + " ' ";*

This code generates a query to authenticate a user who tries to login to a web site. If a malicious user enters " ' OR 1 = 1 -- ' " and " ' ' " instead of a legitimate username and password into their respective fields the query string becomes as follows:

*SELECT * FROM employeeinfo WHERE name = ' ' OR 1=1 -- ' 'AND password = ' ' ' ';*

Any web application that uses this code will be vulnerable to SQLIAs. The character "--" indicates the beginning of a comment, and everything following the comment is ignored.

The database interprets everything after the WHERE token as a conditional statement, and inclusion of the "OR 1=1" clause turns this conditional statement into a tautology which always evaluates to true. Thus, when the above query is executed, the user will bypass the authentication logic and more than one record is returned by the database. As a result, the information about all the users will be displayed by the application, and the attack succeeds. The above discussed example is one of the types of SQLIAs called tautology based SQLIA. Different kinds of SQLIAs known to date are discussed in [6] which include the use of SQL tautologies, illegal queries, union queries, piggy-backed queries, etc.

One of the most widely used techniques to detect and prevent SQLIAs on web applications is input validation. It consists of checking the user input for SQL keywords, such as "FROM", "WHERE", and "SELECT", and SQL operators such as single quotes or comment operator. The rationale behind this technique is that the presence of such keywords and operators may indicate an attempted SQLIA. Input validation technique has not been successful in completely preventing SQLIAs because: 1) the technique is limited by the developer's ability to generate appropriate input validation code and recognize all situations in which it is needed, 2) it results in high rate of false positives, as SQL keywords can be part of a normal text entry and SQL operators can be used to express formulas or even names, and 3) attackers keep finding new attack strings or subtle variations on old attacks that avoid the checks that programmers put in place [7]. This results in anomalous behavior of the application during its execution because of which the attacker gains unauthorized access to the confidential user data.

Software testing, which is one of the important phases in Software Development Life Cycle (SDLC), is performed by developers/testers to assure correctness, quality, security, and reliability of web applications. Traditional functional testing is performed to ensure that software works according to the user specified functional requirements i.e. services the system should provide, outputs that should be displayed by the system for particular inputs, etc [8]. Unfortunately, traditional functional testing cannot fully demonstrate that the software is immune to security attacks like SQLIAs, XSS, path traversal attacks etc., and neither is the best approach to determine the behavior of the software under hostile conditions [9]. This is because: (1) they assume the users of the software are perfect and will never attempt to perform attacks on it, (2) the environment of the software is perfect, as it will never interact with other software that generates hostile return values, and (3) the API or the library functions of the software are perfect [10]. Hence, traditional functional testing should always be augmented with security testing that ensures that the software systems used by organizations and users are secured from unauthorized attacks.

Security testing consists of identification and removal of security vulnerabilities i.e. a defect or weakness in a software system's design, implementation, operation, or management that could be exploited by an attacker. Few of the most commonly used security testing techniques includes automated static analysis, vulnerability scanning, and penetration testing. Automated static analysis [9] involves analyzing the source code of the software without executing it, and is performed using static analysis tools. The main objective of static analysis is to discover security flaws and to identify their potential fixes. The analysis doesn't require knowing what the code is intended to do. Static analysis tools are effective at detecting language rules violations such as buffer overflows, incorrect use of libraries, type checking and other flaws. Static analysis tools have following limitations: 1) inability to detect unexpected flaws – flaw categories must be predefined, 2) inability to detect system administration or user mistakes, and 3) inability to find vulnerabilities introduced by the execution environment.

Automated vulnerability scanning [9] is supported for application level software, as well as web servers, database management systems, and some operating systems. Application vulnerability scanners can be useful for software security testing. These tools scan the executing application software for input and output of known vulnerability patterns, also known as signatures. While they can find simple patterns associated with vulnerabilities, automated vulnerability scanners are unable to pinpoint risks associated with aggregations of vulnerabilities, or to identify vulnerabitlites that result from unpredictable combinations of input and output patterns. Because automated vulnerability scanners are signature-based, they need to be frequently updated with new signatures. In software's target environment, vulnerabilities in software are often masked by environmental protections such as network- and application-level firewalls. Moreover, environment conditions may create novel vulnerabilities that cannot be found by a signature-based tool.

Penetration testing [9] is the "art" of testing a running application in its "live" execution environment to find security vulnerabilities. Penetration testing observes whether the system resists attacks successfully, and how it behaves when it cannot resist an attack. Penetration testers also attempt to exploit vulnerabilities that they have detected and once that were detected in previous reviews. Types of penetration testing include black box, white box, and grey box. In black box penetration testing, the testers are given no knowledge of the application. White box penetration testing is the opposite of black box. In that, complete information about the application may be given to the testers. Grey box penetration testing, the most commonly used, is where the tester is given the same privileges as a normal user to simulate a malicious insider. Many developers use application penetration testing as their primary security testing technique. While it certainly has its place in a testing program, application penetration testing should not be considered the primary or only testing technique. Penetration testing can lead to a false sense of security. Just because an application passes penetration testing doesn't mean that it is free of vulnerabilities. Conversely, if an application fails penetration testing, there is a strong indication that there are serious problems that should be mitigated.

The above mentioned limitations and lack of assurance from security testing of web applications has lead to the exploitation of security vulnerabilities by attackers. This has also enhanced the need for additional tools or methodologies to detect and prevent SQLIAs on web applications. Monitoring an application during runtime determines whether the current execution of the program behaves correctly. There could be some information that is available only at runtime, or the behavior of the system could depend on the environments where the system runs. Also, it is possible that, nevertheless the system has been tested and maybe also proved correct, the developer wants to be sure that system does not violate some given properties during its execution [11]. Hence, runtime monitoring can also be useful to guarantee the security of running programs, and is performed by software runtime monitors.

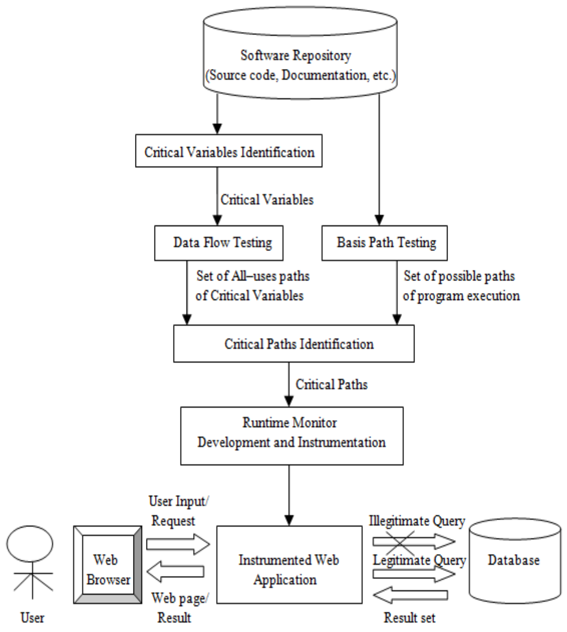
In this paper, we present a framework to counter SQLIAs that uses the idea of runtime monitoring discussed above. The framework first uses pre-deployment testing of web applications to develop and instrument monitors into them. Then, at runtime, the monitors observe the behavior of the application and check them for compliance with the obtained valid/legal execution paths. Any deviation in the application's behavior will be identified as a possible SQLIA, notify the administrator of the attack, and the execution of the application is stopped as an immediate preventive measure. We primarily focus on tautology based SQLIA which is the most popular type of SQLIAs. In this paper, we also present an evaluation of the developed framework on subjects of various types and sizes. We targeted the subject applications with a large number of both legitimate and SQL injection attack inputs, and assessed the ability of the framework to detect and prevent the attacks. The framework was able to detect most of the attacks without generating any false negatives, and successfully allowed all the legitimate inputs to access the database without generating any false positives. Moreover, our technique imposed a low overhead on the subject applications.

II. Runtime Monitoring Framework

In this section, we discuss the proposed Runtime Monitoring Framework to detect and prevent SQLIAs. The framework leverages the artifacts obtained from pre-deployment testing of web applications to develop and instrument runtime monitors, which observe the application's behavior during execution. The key insights behind the development of the framework are that 1) the information essential to identify the possible valid/legal execution paths of a web application can be obtained from the pre-deployment testing i.e. basis path and data flow, and 2) an SQLIA would violate the valid/legal execution paths previously obtained. Therefore, the framework first uses pre-deployment testing of web applications to develop and instrument monitors into them. Then, at runtime, the monitors observe the behavior of the application and check them for compliance with the obtained valid/legal execution paths. Any deviation in the application's behavior will be identified as a possible SQLIA, and its execution is immediately stopped as a preventive measure. The Runtime Monitoring Framework described is shown in Fig. 2 and it consists of the following three phases: 1) Critical Variables Identification, 2) Critical Paths Identification, and 3) Runtime Monitor Development and Instrumentation which are discussed below in detail.

A. *Critical Variables Identification*

Critical variables are those variables that get initialized with the input provided by an external user and are part of a SQL query execution. Since critical variables are the ones that provide an interface between the external world and internal information, it is very important to identify all the critical variables present in the application. We scan the software repository which consists of a collection of documents related to requirements, security specifications, source code, etc., to find all the critical variables present in the source code of the application.


Software Repository (Source code, Documentation, etc.)
Critical Variables Identification
Critical Variables
Data Flow Testing
Basis Path Testing
Set of All–uses paths of Critical Variables
Set of possible paths of program execution
Critical Paths Identification
Critical Paths
Runtime Monitor Development and Instrumentation
User Input/ Request
Web Browser
User
Web page/ Result
Instrumented Web Application
Illegitimate Query
Legitimate Query
Result set
Database
Fig. 2. Runtime monitoring framework.

B. *Critical Paths Identification*

Critical Paths, also called monitorable paths, are the valid/legal execution paths of the application. A combination of two pre-deployment testing techniques i.e. basis path and data flow are used to identify critical paths of the application; runtime monitors developed for the application observes these paths to detect and prevent possible SQLIAs. We first discuss about two pre-deployment testing techniques, basis path and data flow testing techniques, which play a major role in identification of critical paths.

Basis path testing, also known as structured testing, is a methodology for software module testing based on the cyclomatic complexity of McCabe [12], [13] that involves using the source code of a program to find every possible executable path. Functions, procedure, and sub-routines can be called as modules. Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program. The cyclomatic number gives the number of independent paths, called basis paths, through the control flow graph. This means that cyclomatic number is precisely the minimal number of paths that can, in linear combination, generate all possible paths through the application [14]. A control flow graph is an abstract directed graph that describes the control structure of a module. The nodes in the graph correspond to either the computational or conditional statements in a program and the edges represent transfer of control between nodes. As discussed in [12], [13], [15], the four steps devised by McCabe to perform basis path testing are follows: 1) Obtain a control flow graph, 2) Calculate the cyclomatic complexity, 3) Select a basis set of paths, and 4) Generate test cases for each of these paths. Only the first three steps are essential for our framework to help in the development of runtime monitors.

Data flow testing [16] focuses on the variables used within a program and allows the tester to chart the changing values of variables within the program. A typical data flow test case requires that all path segments between a variable being assigned a value and that variable's value being used be covered during testing. With respect to variables, there are two types of nodes as discussed in [17]: defining nodes and usage nodes. For example, with respect to variable x, nodes containing statements such as "input *x*" and "$x = 2$" would be both defining nodes and nodes containing statements such as "print *x*" and "$a = 2 + x$" would be both usage nodes. All-uses (AU) is one of the strategies to perform data flow testing on a variable and this strategy states that: for every use of the variable, there is a path from the definition of that variable to the use [17]. This implies that the paths obtained from data flow testing will be sub-paths of the paths obtained from basis path testing.

Thus, the Critical Paths Identification phase combines the paths obtained from basis path and data flow testing techniques, as discussed above, and builds the set of monitorable paths which are identified using the pseudo code shown below:

Let $C = \{C^1, C^2\ldots\ldots, C^m\}$ be a set of *m* critical variables identified during the Critical Variables Identification Phase.

Let $P_C = \{\{P_C^1\}, \{P_C^2\}, \ldots.., \{P_C^m\}\}$ be a set of critical variable paths such that, P_C^i is a set of paths that a critical variable C^i can take during its lifetime in the software, i [0, m] and is identified by performing data flow testing on C^i.

Let $P = \{P^1, P^2 \ldots\ldots, P^k\}$ be a set of k legal/valid execution paths identified using basis path testing and CP is a set of monitorable paths.

```
$CP = \{ \}$
for every $P^j$  $P$ and
    for every $P_C^i$  $P_C$
        if $(P^j \cap P_C^i == P_C^i)$
            $CP = CP \cup \{P^j\}$
$i$  $[0, m]$ and $j$  $[0, k]$
```

C. Runtime Monitor Development and Instrumentation

A runtime monitor is developed for the execution paths of the application obtained from the Critical Paths Identification phase. Execution paths of the program are monitored because of the following reasons as discussed in [18]: 1) paths offer insights into a program's dynamic/runtime behavior that is difficult to achieve any other way and 2) paths capture some of the usually invisible dynamic sequencing of statements and records a program's executable statements in the order in which they run. Events occur instantaneously during a program execution and consist of variable updates, method calls and returns, etc. A program's sequence of events over an execution is a rich source of information about the program's behavior on that execution. Runtime monitoring can detect such sequence of events, enabling developers to handle the sequences with reporting or recovery code [19].

Dynamic events are mapped onto events in the actual code base. One potentially good way of doing this would be to use pointcuts of an aspect oriented programming language [20]. As discussed in [21], [22], pointcuts define a collection of specific points in the dynamic execution of the application. On pointcuts, advice can then be defined in order to execute certain code. AspectJ supports before and after advice, depending on the time the code is executed. The definition of pointcuts and the specification of advice on these pointcuts together form an aspect definition. Such pointcuts have proven themselves to be easy enough to understand for many average software developers, as indicated by their wide-spread use in software development.

Runtime Monitors are developed using AspectJ [23] which is an aspect-oriented extension to the Java language and follows the Aspect Oriented Programming (AOP) paradigm. AOP has been widely used in all areas of software development and recently it is being widely used in the area of software security. AOP builds on pervious technologies such as procedural-oriented and object-oriented programming. With AOP, a programmer could do some of the following to achieve software security: (1) write aspects to address security concerns such as secrecy and integrity [24], (2) write an aspect to define a set of sensitive operations that says "before each sensitive operation, check the user's access level" [25], etc. Thus, AOP provides specific language mechanisms that make it possible to address concerns, such as security in a modular way. This way, the security issue in a software system can be addressed [26].

A special complier provided by AspectJ called the AspectJ Compiler (AJC) is used to instrument monitors into the respective modules of the application. During runtime, if the path taken by the application violates the valid/legal execution paths obtained, this implies that the input from the external user is malicious, and the query formed is trying to access the confidential user data. This abnormal behavior of the application is detected by the instrumented runtime monitor as a possible SQLIA, notifies the administrator of the attack, and the execution of the application is stopped as an immediate preventive measure.

III. Evaluation

The goal of the evaluation is to assess the effectiveness and efficiency of the Runtime Monitoring Framework, presented in this paper, to detect and prevent tautology based SQLIA when applied to various web applications. We investigate the following three research questions:

RQ1: Does Runtime Monitoring Framework detect and prevent tautology based SQLIA that would otherwise remain undetected? (False Negatives)

RQ2: Does Runtime Monitoring Framework detect legitimate inputs as tautology based SQLIA and prevent their execution on the database? (False Positives)

RQ3: What is the runtime overhead imposed on the instrumented web application by the Runtime Monitoring Framework?

The rest of this section is organized as follows. First, we illustrate the setup for our evaluation, i.e. SQL Injection Application Testbed that consists of web applications, and Target/Subject web applications that we used for performing our experiments. We then describe the attack and legitimate test input data collected and the results obtained for each of the research questions listed above. Finally, we compare our approach with other techniques and discuss the observations made.

TABLE I: ILLEGITIMATE TAUTOLOGY BASED ATTACK INPUTS

Illegitimate inputs	
Login	Password
' OR 1=1 -- '	' '
' '	' OR '1'='1
' 'aaa' OR 1=1 -- '	' '
' '111' OR 1=1 -- '	' '
' '333' OR true#'	' '
' 'admin' OR 1<2 -- '	' '
' 'login' OR 4>2 -- '	' '
' '	' OR 1=1 -- '
' '	' 'bbb' OR 1=1 -- '
' '	' '222' OR 1=1 -- '
' '	' '444' OR true#'
' '	' 'password' OR 4>2 -- '
' '	' 'admin' OR 10<100 -- '
' '	' or user_password between 'a' and 'z
' or user_login between 'a' and 'z	' 'password' OR 4>2 -- '
' or user_login between 'a' and 'z	' '
' OR '1'='1--	' '

A. *Target/Subject Applications Used for Evaluation*

SQL Injection Application Testbed [27] provides a set of subject web applications that are vulnerable to SQLIAs. The testbed was developed to facilitate the evaluation of various techniques and methodologies to detect and prevent SQLIAs. The set of subjects consists of seven web applications that are vulnerable to SQLIAs, which accept user input via forms and use the input to build queries to gain access to the underlying database. The first five subjects in the testbed i.e. Classifieds, Events, Employee Directory, Bookstore, and Portal are commercial web applications, and the remaining two applications in the testbed, Checkers and Office Talk, are developed by students and have been used in other related works in [28]. For the purpose of experimentation, we chose the first three applications i.e. Events, Classifieds, and Employee Directory from the SQL Injection Application Testbed to evaluate the framework. Events application is an online tracking system, which can be used to schedule various events in an organization, and is developed using Java Server Pages (JSPs). The users can only view information of events scheduled whereas the administrator has the privilege to add, remove, and edit information related to scheduled events. Classifieds application is an online management system developed using JSPs. Employee Directory application is an online management system developed using JSPs. This application can be used in an organization to look up for employee details such as name, email, department, etc. The administrator of the application has the privilege to add, remove, and edit employee or department related information.

B. *Test Input Data Collection*

For our evaluation we collected a large set of inputs by surveying various sources which included government security websites such as NVD (http://www.nvd.nist.gov/), OWASP (https://owasp.org/), Build Security In (https://buildsecurityin.us-cert.gov/), US-CERT (http://www.us-cert.gov/), security related mailing lists, research papers, etc. The generated inputs represented both malicious and normal usage of the target/subject applications. Illegitimate tautology based attack inputs, listed below in Table I, consist of statements that are inherently true and are used to force a query to return all results, ignoring any WHERE conditionals.

Legitimate inputs, listed below in Table II, consist of SQL keywords, operators, and troublesome characters, such as single quotes and comment operators, but in a way that should not cause an attack.

TABLE II: LEGITIMATE INPUTS

Legitimate inputs	
Login	Password
test	****
select * from tab	****
insert into user values(\’test\’,\’test\’)	****
union select * from tab	****
insert	****
delete	****
from	****
where	****
group by	****
left join	****
create	****
right outer join	****
procedure	****
information_schema	****
view	****
like	****
and	****
or	****
!@#$$%^&*()*_!%2B\\\\\\:;[]{}><, .	****
-%2B=_	****
`~	****
QWERTYUIOP	****
ASDFGHJKL	****
ZXCVBNM<>?	****
< >	****
////////	****
UNION	****
Update Set	****
test@localhost.com	****
sec45%^	****
rdharam	*************
bobk	****
johns	*******
davids	****
pabols	************

TABLE III: RESULTS OBTAINED WHEN ILLEGITIMATE TAUTOLOGY BASED ATTACK INPUTS ARE PROVIDED TO THE INSTRUMENTED WEB APPLICATIONS

Illegitimate tautology based attack inputs		Attack Detected
Login	Password	
' '	' OR '1'='1	YES
' OR 1=1 -- '	' '	YES
' 'aaa' OR 1=1 -- '	' '	YES
' '111' OR 1=1 -- '	' '	YES
' '333' OR true#'	' '	YES
' 'admin' OR 1<2 -- '	' '	YES
' 'login' OR 4>2 -- '	' '	YES
' '	' OR 1=1 -- '	YES
' '	' 'bbb' OR 1=1 -- '	YES
' '	' '222' OR 1=1 -- '	YES
' '	' '444' OR true#'	YES
' '	' 'password' OR 4>2 -- '	YES
' '	' 'admin' OR 10<100 -- '	YES
' '	' or user_password between 'a' and 'z	YES
' or user_login between 'a' and 'z	' 'password' OR 4>2 -- '	YES

TABLE IV: RESULTS OF TESTING FOR FALSE NEGATIVES (RQ1)

Target/Subject application	Total # of tautology based attack inputs	Total # of attacks detected on instrumented web application
Events	15	15
Classifieds	15	15
Employee Directory	15	15

TABLE V: RESULTS OBTAINED WHEN LEGITIMATE INPUTS ARE PROVIDED TO THE INSTRUMENTED WEB APPLICATIONS

Legitimate inputs		Query Successful
Login	Password	
test	****	YES
select * from tab	****	YES
insert into user values(\'test\',\'test\')	****	YES
union select * from tab	****	YES
insert	****	YES
delete	****	YES
from	****	YES
where	****	YES
group by	****	YES
left join	****	YES
create	****	YES
right outer join	****	YES
procedure	****	YES
information_schema	****	YES
view	****	YES
like	****	YES
and	****	YES
or	****	YES
!@#$$%^&*()*_!%2B\\\\\\:;[ ]{}><,.	****	YES
-%2B=_	****	YES
`~	****	YES
QWERTYUIOP	****	YES
ASDFGHJKL	****	YES
ZXCVBNM<>?	****	YES
< >	****	YES
////////	****	YES
UNION	****	YES
Update Set	****	YES
test@localhost.com	****	YES
sec45%^	****	YES
rdharam	*************	YES
bobk	****	YES
johns	*******	YES
davids	****	YES
pabols	************	YES

C. Discussion of Results

In this section, we discuss the results obtained when the runtime monitor is instrumented into target/subject web applications i.e. Events, Classifieds, and Employee Directory respectively. To address RQ1 (i.e. the effectiveness of the Runtime Monitoring Framework to detect and prevent tautology based SQLIA), we instrumented each target/subject application with runtime monitors and ran all of the attack inputs listed in Table I. For every application, we observed whether the tautology based SQLIA was detected and prevented by the runtime monitor. (As previously discussed, when the runtime monitor detects an attack, it stops the execution of the web application. Therefore, it is easy to accurately detect the occurrence of an attack.) The results obtained when illegitimate tautology based attack inputs (*Login* and *Password*) are provided to the instrumented subject/target applications are shown in Table III. The *Attack Detected* column in the table will have "YES" value if the attack is detected successfully by the runtime monitor, else it contains a "NO" value. As the table shows, for all subjects the runtime monitor was able to correctly identify all attacks as tautology based SQLIA, that is, it generated no false negatives.

As shown in Table IV, the runtime monitor successfully detected all illegitimate tautology based SQLIA performed on the target/subject web applications i.e. Events, Classifieds, and Employee Directory.

To address RQ2 i.e. Does Runtime Monitoring Framework identify legitimate accesses as tautology based SQLIA and prevent from executing on the database? (False Positive), we ran legitimate inputs listed in Table II on the instrumented target/subject web applications, and assessed whether runtime monitor identified any legitimate query as an attack. Table V summarizes the results obtained when legitimate inputs are provided to the instrumented web applications. The *Query Successful* column will have a "YES" value in case of a successful query execution, else it contains a "NO" value. The results of the assessment were that runtime monitor correctly identified all such queries as legitimate queries and reported no false positives.

As shown below in Table VI, the monitor successfully allowed the legitimate queries to be executed on the target web applications without falsely detecting them as attack.

TABLE VI: RESULTS OF TESTING FOR FALSE POSITIVES (RQ2)

Target/Subject application	Total # of legitimate inputs	Total # of legitimate inputs detected as attacks on instrumented web application
Events	35	0
Classifieds	35	0
Employee Directory	35	0

To address research question 3 (RQ3) i.e. what is the runtime overhead imposed on the instrumented web application by Runtime Monitoring Framework? (Runtime Overhead) we ran the legitimate inputs on the uninstrumented target/subject web applications and measured its response time. We then ran the same legitimate inputs on the instrumented version of the subject applications and recorded the response time. The difference in the response time obtained from the two versions of the application is determined as the overhead imposed by the framework. We performed our experiments five times and recorded the average time to ensure accuracy. Only the legitimate test input data is used for overhead calculation, because using the attack set would cause different paths of execution between the two versions, where the attacks would be successful in the original application, but be prevented in the instrumented application, leading to incorrect timing comparisons [29].

We compared our results with recently proposed work, CANDID [30] and WASP [31]. CANDID automatically transforms web applications to render them safe against all SQLIAs. It dynamically mines the programmer-intended query structure on any input and detects attacks by comparing it against the structure of the actual query issued. WASP is based on dynamic tainting and has been widely used to address security problems related to input validation. Traditional dynamic tainting approaches mark untrusted data from user input as tainted, track the flow of tainted data at runtime, and prevent this data from being used in potentially harmful ways. We found that the runtime overhead imposed by the Runtime Monitoring Framework on target applications is no more than 4% which is comparatively less than the average overhead of WASP listed as 6% and CANDID which is 6.2%. Table VII below shows the comparison results obtained.

TABLE VII: COMPARISON WITH OTHER TECHNIQUES

Technique	Runtime Overhead
Runtime Monitoring Framework	4%
CANDID	6.2%
WASP	6%

In this section, we discussed the evaluation and results obtained to assess the effectiveness and efficiency of Runtime Monitoring Framework. We addressed the following three research questions: False Negatives (RQ1), False Positives (RQ2), and Runtime Overhead (RQ3). We also compared our approach with other techniques i.e. CANDID and WASP. The framework successfully allowed the legitimate inputs to be executed on the target/subject web applications; furthermore, it detected illegitimate attack inputs. During runtime, the instrumented monitors observed the behavior of the application for every given user input and detected a possible SQLIA in case of an attack input; the execution of the application was then stopped as an immediate preventive measure. Thus, based on the behavior of the application during runtime, SQLIAs were effectively detected and prevented by the instrumented monitors. Also, the results of the evaluation performed clearly demonstrated the success of the framework to detect and prevent tautology based SQLIAs. The framework also imposed a low runtime overhead on the target applications.

IV. RELATED WORK

This section discusses the related work that has been accomplished by the research community in providing new techniques to detect and prevent SQLIAs. We also discuss state-of-the-art of SQLIA detection and prevention techniques and classify them into two categories namely: (A) Pre-deployment Techniques and (B) Post-deployment Techniques.

A. Pre-Deployment Techniques

Pre-deployment techniques consist of methodologies that are used to detect SQLIAs and their vulnerabilities during coding time and testing time of an application development cycle. Static analysis techniques detect SQLIAs and their vulnerabilities during coding time without the need of code execution. Different testing approaches such as black-box testing and white-box testing can be used as analysis methods in testing time for detecting attacks and their vulnerabilities [32]. In this section, we discuss the techniques that can be categorized as pre-deployment and compare them with our approach.

Huang *et al.*, [33] proposed WAVES, a black box technique for testing web applications for SQL Injection

Attacks. The technique identifies all points in a web application that can be used to inject SQLIAs using a web crawler. It then builds attacks that target those spots based on a list of patterns, and then monitors the application's response to the attacks by utilizing machine learning to improve its attack methodology.

Wasserman and Su [34] proposed a static analysis framework that operates directly on the source code of the application to detect and prevent SQLIAs. This approach consists of two main steps. First, static analysis is performed to approximate the set of possible queries that the program generates for a particular query variable at a particular program location. The result for each query variable is a finite state automaton which represents a conservative set of possible string values that the variable can take. In the second step, the part of the generated automaton corresponding to the WHERE clause of the generated queries are analyzed to check whether there is a tautology, and the existence of a tautology indicates the presence of a potential vulnerability.

Livshits and Lam [35] discuss a static analysis technique to detect SQL injection vulnerabilities in web applications. In this approach, users describe vulnerability patterns of interest using Program Query Language (PQL) which is an easy-to-use language with Java-like syntax. The user-provided specifications of vulnerabilities are then automatically translated into static analyzers which find all potential matches of vulnerabilities in the code statically. The advantage of static analysis is that it can find all potential security violations without executing the application. The primary limitation of this approach is that it can only detect known and specified vulnerability patterns of SQLIAs and cannot detect SQL injection attacks patterns that are not known beforehand. In our approach no user-defined specifications are used and SQLIAs are detected based on the behavior of the application.

Kosuga *et al.*, [36] proposed Sania designed to be used in the development and debugging phase of web applications to detect SQL injection vulnerabilities. To discover SQL injection vulnerabilities, Sania analyzes SQL queries issued in response to the HTTP requests between web application and database, and discovers vulnerable spots in SQL queries in which an attacker can insert malicious strings. It then generates attack requests based on the context of potentially vulnerable spots in the SQL queries. Parse trees of the SQL queries are generated and compared.

All the above mentioned techniques are used to detect and prevent SQLIAs in web application during its pre-deployment i.e. either during coding time or testing time of the application development cycle. Thus, in spite of the existence and the implementation of above discussed pre-deployment techniques, web applications are still vulnerable to SQLIAs because hackers are successfully able to circumvent the employed techniques. Hence detecting and preventing SQLIAs on web applications after it is deployed in the real world i.e. post-deployment technique is essential.

B. Post-Deployment Techniques

Post-deployment techniques consist of methodologies that are used to detect SQLIAs and their vulnerabilities during operation time i.e. in the real world field after the product is released [32]. In this section, we discuss the methodologies that can be categorized as post-deployment, and compare them with our approach.

Valeur *et al.*, [37] proposed an Intrusion Detection System (IDS) based on a machine learning technique to detect SQLIAs. The proposed system uses anomaly-based detection approach and learns profiles using a number of different models to find the normal database access performed by web applications. During the training phase, profiles are learned automatically by analyzing a number of sample database accesses. During the detection phase, anomalous queries that lead to SQLIA are identified.

Kemalis and Tzouramanis [38] proposed a novel specification-based methodology SQL-IDS for the detection of exploitation of SQL injection vulnerabilities. This approach focuses on writing specifications that describe the intended structure of SQL queries that are produced by the web application. It then automatically monitors the execution of the application for the SQL queries that violate the pre-defined query specification rules. The approach detects and prevents the application from all forms of SQLIAs and also eliminates the need to modify the source code of the application. SQL-IDS incur high computation cost while comparing the new query with the predefined structure at runtime.

Halfond and Orso [39] proposed a model-based technique called AMNESIA for detection and prevention of SQLIAs that combines the static and dynamic analysis. During the static phase, models for the different types of queries that an application can legally generate at each point of access to the database are built. During the dynamic phase, queries are intercepted before they are sent to the database and are checked against the statically built models. If the queries violate the model then a SQLIA is detected and further queries are prevented from accessing the database. The accuracy of AMNESIA depends on the static analysis for building query models.

Buehrer *et al.*, [40] presented a novel runtime technique to eliminate SQL injection. The technique is based on comparing at runtime the parse tree of the SQL statement before inclusion of user input with that resulting after inclusion of input. SQLGuard requires the application developer to rewrite code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query. SQLGuard uses a secret key to delimit user input during parsing by the runtime checker and so the security of the approach is dependent on the attacker not being able to discover the key.

Haldar *et al.*, [41] proposed a framework called Java Dynamic Tainting for tagging, tracking and detecting the improper use of improperly validated user input also called tainted input in web applications. The data originated from the client is marked as tainted, and this attribute is propagated throughout the execution of the program. Tainted flag is associated with strings and data originating from methods that get user input, called sources, that are marked tainted. Strings derived from tainted strings are also marked tainted. Finally, methods that consume input or execute some form of code (scripts, SQL), called sinks, are prevented from taking in tainted arguments.

Boyd and Keromytis [42] proposed SQLrand, which is an approach based on instruction-set randomization. The standard SQL keywords in queries are modified by appending a random integer value during the design time of the application. During runtime, a proxy that sits between the client and the database server intercepts the SQL queries and de-randomizes the query by removing the inserted random integer before submitting the queries to the database. Therefore, any malicious user attempting an SQLIA will not be successful as the user input inserted into the randomized query will be classified as a set of non-keywords resulting in an invalid expression.

Bisht *et al.*, [30] exhibit a novel and powerful mechanism called CANDID for automatically transforming web applications to render them safe against all SQLIAs. The proposed technique dynamically mines the programmer-intended query structure on any input and detects attacks by comparing it against the structure of the actual query issued.

Halfond *et al.*, [31] proposed a highly automated approach for dynamic detection and prevention of SQLIAs. The approach is based on dynamic tainting which has been widely used to address security problems related to input validation. Traditional dynamic tainting approaches mark untrusted data from user input as tainted, track the flow of tainted data at runtime, and prevent this data from being used in potentially harmful ways. Unlike any existing dynamic tainting techniques, the proposed approach is based on the novel concept of positive tainting i.e. identification and marking of trusted instead of untrusted data. The proposed approach performs accurate taint propagation by precisely tracking trust markings at the character level, and it performs syntax-aware evaluation of query strings before they are sent to the database and blocks all queries whose non-literal parts (i.e. SQL keywords and operators) contain one or more characters without trust markings.

All the above mentioned techniques are used to detect SQLIAs and their vulnerabilities during its post-deployment (operation time) i.e. in the real world field after the product is released. Machine Learning Techniques that includes IDS, and SQL-IDS discussed above are mainly dependent on the accuracy of the profiles obtained during the training phase, and it is possible that a few of the SQLIAs may go unnoticed causing threat to the database. In order to overcome this limitation, in our approach we monitor the legitimate behavior of the application during its execution and eliminate the need for a training data set to detect and prevent SQLIAs. Java Dynamic Tainting requires modifications to the runtime environments which in turn affects their portability. In our approach, we embed a runtime monitor into the relevant module of the application that monitors its behavior for a given user input, and no change to the runtime environment is required to detect and prevent SQLIAs. AMNESIA and SQL Guard construct syntactic models like parse trees and FSA by using static analysis technique to identify the intended structure of SQL queries in the absence of user inputs; these approaches then use dynamic analysis, and detect SQLIAs at runtime if the dynamically generated query, which includes user inputs, deviates from the statically generated syntactic models. In our proposed approach, pre-deployment testing techniques are used to find the valid/legal behaviors of the application in the presence of user input. During runtime, the developed monitors observe the execution of the application and any deviation is determined as a possible SQLIA. Instruction Set Randomization techniques such as SQL rand requires developers to randomize SQL queries present in the application by appending a random integer value. In our proposed approach, SQL queries are written using standard keywords, and runtime monitors are developed and instrumented into the source code automatically. Also, the need for the deployment of proxy is eliminated.

Based on the above analysis and limitations of various pre-deployment and post-deployment techniques, discussed in this section, we designed a framework that utilizes the artifacts obtained from pre-deployment testing of applications for the development and instrumentation of runtime monitors. Thus, SQLIAs are effectively detected and prevented based on the behavior of the application during its execution.

V. Conclusion

In this paper, we presented a Runtime Monitoring Framework to detect and prevent SQL Injection Attacks. The framework first uses pre-deployment testing of web applications to develop and instrument monitors into them. Then, at runtime, the monitors observe the behavior of the application and check it for compliance with the obtained valid/legal execution paths. Any deviation in the application's behavior will be identified as a possible SQLIA, notify the administrator of the attack, and the execution of the application is stopped as an immediate preventive measure. We also presented an evaluation of the framework performed on a set of web applications from SQL Injection Application Testbed. We targeted the subjects with a large number of both legitimate and SQL injection attack inputs, and assessed the ability of the framework to detect and prevent the attacks without stopping any legitimate accesses to the database. The framework was able to detect most of the attacks without generating any false negatives, and successfully allowed all the legitimate inputs to access the database without generating any false positives. Moreover, our technique imposed a low overhead on the subject applications compared to other techniques. Thus, using our framework, we ensure that the quality and security of web application is achieved not only during its pre-deployment, but also during its post-deployment phase. We aim to extend our work to detect and prevent other types of SQLIAs.

References

[1] Silobreaker. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10

[2] Sony Hacked Again in Lulzsec Breach. [Online]. Available: http://www.zdnet.com/sony-hacked-again-in-lulzsec-breach-4010022607/

[3] Zdnet. [Online]. Available: http://www.zdnet.com/blog/btl/6-46-million-linkedin-passwords-leaked-online/79290

[4] Nvidia Confirms Hackers Swiped Up to 400,000 User Accounts. [Online]. Available: http://www.zdnet.com/nvidia-confirms-hackers-swiped-up-to-400000-user-accounts-7000000903/

[5] 8.24 Million Gamigo Passwords Leaked After Hack. [Online]. Available: http://www.zdnet.com/8-24-million-gamigo-passwords-leaked-after-hack-7000001403/

[6] G. J. Halfond, J. Viegas, and A. Orso, "A classification of SQL Injection Attacks and countermeasures," in *Proc. the IEEE International Symposium on Secure Software Engineering*, pp. 13-15, 2006.

[7] W. G. Halfond and A. Orso, "AMNESIA: analysis and monitoring for neutralizing SQL Injection Attacks," in *Proc. the 20th IEEE/ACM*

International Conference on Automated Software Engineering, pp. 174-183, 2005.

[8] I. Sommerville, *Software Engineering*, Addison Wesley, 2001.

[9] Build Security In, “Software security is testing,” *Software Assurance Pocket Guide Series: Development*, vol. III, May 21, 2012.

[10] A. J. A. Wang, “Security testing in software engineering courses,” *34th ASEE/IEEE Frontiers in Education*, pp. F1C-13, 2004.

[11] P. Arcaini, A. Gargantini, and E. Riccobene, *Runtime Monitoring of Java Programs by Abstract State Machine*s, TR 131, DTI Dept., Univ. of Milan, 2010.

[12] T. J. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, 1976.

[13] T. J. McCabe and A. H. Watson, “Structured testing: a software Testing methodology using the cyclomatic complexity metric,” NIST Special Publication 500-235, 1996.

[14] Y. Deng and J. Wang, “Testing web database applications,” *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 5, pp. 1-10, 2004.

[15] L. Gregory, H. Schligloff, and M. Roggenbach, “Path testing, Advance topics in computer science,” Testing, Swansea University, Wales, UK, 2002.

[16] M. New, “Data flow testing,” Advance Topics in Computer Science, Swansea University, Wales, UK, 2002, 2012.

[17] J. Badlaney, R. Ghatol, and R. Jadhwani, “An introduction to data-flow testing,” Department of Computer Science, North Carolina State University, NCSU CSC TR-2006-22, 2006.

[18] T. Ball and J. R. Larus, “Using paths to measure, explain, and enhance program behavior,” *IEEE Computer*, vol. 33, no. 7, pp. 57-65, 2000.

[19] E. Bodden, P. Lam, and L. Hendren, “Finding programming errors earlier by evaluating runtime monitors ahead-of-time,” in *Proc. the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ACM, pp. 36-47, 2008.

[20] E. Bodden, “The design and implementation of formal monitoring techniques,” in *Proc. 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, ACM, pp. 939-940, 2007.

[21] D. Walker, S. Zdancewic, and J. Ligatti, "A theory of aspects," in *ACM SIGPLAN Notices*, vol. 38, no. 9, pp. 127-139, 2003.

[22] K. Padayachee and N. Wakaba, "A taxonomy of aspect-oriented security," *Review of Business Information Systems (RBIS)*, vol. 12, no. 1, pp. 89-102, 2011.

[23] R. Miles, *AspectJ Cookbook*, O’ Reilly, December 27, 2004.

[24] J. Viega, J. T. Bloch, and P. Chandra, "Applying aspect-oriented programming to security,” *Cutter IT Journal*, vol. 14, no. 2, pp. 31-39, 2001.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with aspect, "*Communications of the ACM*, vol. 44, no. 10, pp. 59-65, 2001.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An overview of aspect,” *ECOOP 2001- Object-Oriented Programming*, pp. 327-354, 2001.

[27] BCF. [Online]. Available: http://www-bcf.usc.edu/~halfond/testbed.html

[28] Z. Su and G. Wassermann, “The essence of command injection attacks in web Applications,” *ACM SIGPLAN Notices*, vol. 41, no. 1, pp. 372-382, 2006.

[29] R. Mui and P. Frankl, “Preventing SQL injection through automatic query sanitization with ASSIST,” in *Proc. Fourth International Workshop on Testing, Analysis and Verification of Web Software*, vol. 35, pp. 27-38, 2010.

[30] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, “CANDID: dynamic candidate evaluations for automatic prevention of SQL Injection Attacks,” *ACM Transactions on Information and System Security*, vol. 13, no. 2, 2010.

[31] W. G. J Halfond, A. Orso, and P. Manolios, “WASP: protecting web applications using positive tainting and syntax-aware evaluation,"*IEEE Transaction on Software Engineering*, vol. 34, no.1, pp. 65-81, 2008.

[32] A. Shakya and D. Aryal, "A Taxonomy of SQL injection defense techniques," M. S. thesis, School of Computing, Blekinge Institute of Technology, Karlskrona, Sweden, 2011.

[33] Y.W. Huang, S. K. Huang, T. P. Lin, and C. H. Tsai, “Web application security assessment by fault injection and behavior monitoring,” in *Proc. the 12th International Conference on World Wide Web*, ACM, pp. 148-159, 2003.

[34] G. Wassermann and Z. Su, “An analysis framework for security in web applications,” in *Proc. the FSE Workshop on Specification and Verification of Component Based Systems*, pp. 70-78, 2004.

[35] V. B. Livshits and M. S. Lam, “Finding security errors in Java programs with static analysis,” in *Proc. the 14th Usenix Security Symposium*, vol. 14, pp. 18-18, 2005.

[36] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama “Sania: syntactic and semantic analysis for automated testing against SQL injection,” in *Proc.23rd Annual Computer Security Applications Conference*, IEEE, pp. 107-117, 2007.

[37] F. Valeur, D. Mutz, and G. Vigna, “A learning based approach to the detection of SQL attacks,” in *Proc. the Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, pp. 123-140, 2005.

[38] K. Kemalis and T. Tzouramanis, “SQL-IDS: A specification-based approach for SQL injection detection,” in *Proc. the ACM Symposium on Applied Computing*, pp. 2153-2158, 2008.

[39] W. G. J. Halfond and A. Orso, “Combining static analysis and runtime monitoring to counter SQL Injection Attacks,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1-7, 2005.

[40] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti, “Using parse tree validation to prevent SQL Injection Attacks,” in *Proc. the 5th International Workshop on Software Engineering and Middleware*, ACM, pp. 106-113, 2005.

[41] V. Haldar, D. Chandra and M. Franz, “Dynamic taint propagation for Java,” in *Proc. 21st Annual Computer Security Applications Conference*, IEEE, 2005.

[42] S. W. Boyd and A. D. Keromytis, “SQLrand: preventing SQL Injection Attacks,” in *Proc. the 2nd Applied Cryptography and Network Security Conference*, Springer Berlin, Heidelberg, pp. 292-302, 2004.

Ramya Dharam is a Ph.D. candidate in the Department of Computer Science at the University of Memphis. She received Ms in computer engineering from Santa Clara University, USA in 2009, B. E. in computer science and engineering from Vishveshwariah Technological University, Bangalore, India in 2007. Her research interests include software engineering, software/security testing, cyber security, and cloud computing.

Sajjan G. Shiva is a professor and the chair of Department of Computer Science at the University of Memphis. He received Ph.D. and M.E.E. degrees in electrical engineering from Auburn University, USA in 1975 and 1971, and B.E. electrical engineering, from Bangalore University, India in 1969.

He has served on the Computer Science Faculty at the University of Alabama in Huntsville and Alabama A&M University. He has also served as the manager of Software Quality Assurance at Teledyne Brown Engineering; Senior Software Engineer and Executive Manager (Technical) at Intergraph Corporation; and Technical Advisor, Computer Technologies Division, U.S. Army Space and Strategic Defense Command. Dr. Shiva has consulted with industry and government organizations in the areas of software engineering, computer architecture, artificial intelligence, and expert systems. He has authored a series of four books on Computer Design and Architecture.

Dr. Shiva is a fellow of IEEE and a life member of ACM. He has received research funding from NSF, NASA, U.S. Department of Defense, and ONR. His current interests are game theoretic cyber security and secure software engineering.