underlying web applications to retrieve, modify and delete confidential user information that are stored in the database resulting in security violations, identity theft, etc. SQLIAs occur when data provided by the user is included directly in a SQL query and is not properly validated. Attackers take advantage of this improper input validation and submit input strings that contain specially encoded database commands [2]. Different kinds of SQLIAs known to date are discussed in [3, 4] which include the use of SQL tautologies, illegal queries, union query, piggy-backed queries, etc.

Even though the vulnerabilities leading to SQLIAs are well understood, the attack continues to be a problem due to lack of effective techniques for detecting and preventing them. In spite of improved coding practices to theoretically prevent SQLIAs, techniques such as defensive programming have been less effective in addressing the problem. Furthermore, attackers continue to find new exploits to circumvent the input checks used by programmers [3]. Software Testing techniques specifically designed to target SQLIAs provide partial solutions to the problem due to following reasons: Firstly, web applications have a very short time-to-market, and hence developers often tend to neglect the testing process; secondly, it is considered too time consuming, expensive, and difficult to perform complete testing of the software [5], and thirdly, testing does not guarantee that all possible behaviors of the implementation are explored, analyzed, and tested [6]. This lack of assurance from testing of web applications has lead to the exploitation of security vulnerabilities by attackers to perform attacks such as SQLIAs.

Pre-deployment testing techniques, such as static analysis, source code review, etc., perform security tests with the software before they are deployed in its actual target environment. These techniques are either too imprecise or focus only on a specific aspect of the problem [7]. Post-deployment testing techniques, such as vulnerability scanning and penetration testing perform security tests with the software deployed in its actual target environment [8]. These techniques are either signature based, or often suffer from issues related to completeness that sometimes result in false negatives being produced [8].

In this paper, we introduce a framework called Runtime Monitoring Framework that is used by our technique to handle tautology based SQLIAs. The framework uses knowledge gained from pre-deployment testing of web application to develop runtime monitors which perform post-deployment monitoring of web application. Basis-path and data-flow testing are the two pre-deployment testing techniques used by the framework to initially find a minimal set of legal/valid execution paths of the application. Runtime monitors are then developed for the identified paths and integrated into the application. The integrated monitors observe the behavior of the application for the valid/legal paths during its post-deployment, and any deviation will be immediately identified as the possible occurrence of tautology based SQLIAs. The monitor then halts the execution of the application and notifies the administrator about the attack.

In this paper, we also present preliminary evaluation of our proposed technique. We implemented the technique on a target web application. The application was provided with a huge set of legitimate and illegitimate inputs. The results obtained were promising as the runtime monitor developed for the subject was able to handle all of the tautology based SQLIAs.

The rest of our paper is organized as follows. In Section 2, we discuss our research strategy and methodology. In Section 3, we discuss the implementation of our proposed framework. Evaluation and results obtained are discussed in Section 4. We discuss related work in Section 5 and conclude in Section 6 with a discussion of future work.

2 RESEARCH STRATEGY AND METHODOLOGY

In this section, our research strategy and methodology to design Runtime Monitoring Framework is discussed. The main idea of our work is to check if the current behavior of the application satisfies the specified behavior; any deviation in the behavior will be immediately detected as the possible exploitation of SQL injection vulnerability. Our framework uses the information gathered from pre-deployment testing of web application to help in the development of runtime monitor.

2.1 Modeling Tautology based SQL Injection Attacks

A Web application structure is shown below in the Figure 1. The three-tiered architecture consists of a web browser, an application server, and a backend database server. A Web application with such an architecture construct database queries dynamically with the received user input and dispatch the queries over an application programming interface (API) to the underlying database for execution.

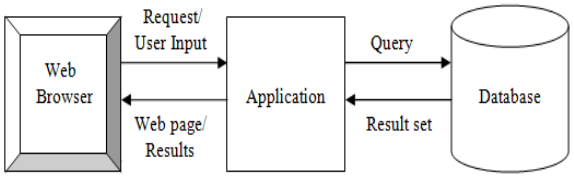

Figure 1: Web Application Structure

The application will then retrieve and present data to the user based on the user's input. Serious security problems arise if the user's inputs are not handled properly. In particular, SQLIAs occurs when a malicious user passes crafted input as part of the query, causing the web application to generate and send a query that in turn results in unintended behavior of the application. This causes the loss of confidential user information.

For example, if a database contains usernames and passwords, the application may contain code such as the following:

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

This code generates a query intended to be used 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 website that uses this code would be vulnerable to tautology based 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 the inclusion of “OR 1=1” clause turns this conditional into a tautology whose condition 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.

2.2 Proposed Framework

The basic idea of our proposed framework is the usage of information gathered from pre-deployment testing of web application, to help in development of runtime monitor to detect and prevent tautology based SQLIAs.

Our proposed framework first uses a software repository which consists of a collection of documents related to requirements, security specifications, source code, etc., to find the critical variables. A Combination of basis-path and data-flow testing techniques is then used to find all the legal/valid execution paths that the critical variables can take during their lifetime in the application. Data-flow analysis testing [10] is an effective approach to detect improper use of data and can be performed either statically or dynamically. In static data-flow analysis, the source code is inspected to track the sequences of uses of data items without its execution. However, in the dynamic data-flow analysis, the sequences of actions are tracked during execution of the program. In our proposed framework we use static data-flow analysis. Basis-path testing is a white box testing technique that identifies the minimal set of all legal execution paths [11] from both the control flow graph of the program, and by the calculation of cyclomatic complexity - the measure of number of independent paths in the program being considered. We thus make use of the aforementioned pre-deployment testing techniques, i.e. basis-path and data-flow techniques, to identify the minimum number of critical paths to be monitored during the post-deployment phase of the application.

Runtime monitor is then developed to observe the path taken by critical variables and check them for compliance with the obtained legal paths. During runtime, if the path taken by the identified critical variables violates the legal paths obtained, this implies that the critical variables consist of the malicious input from the external user and the query formed is trying to access confidential information from the back-end database. This abnormal behavior of the application, due to the critical variables, is identified by the runtime monitor and immediately notified to the administrator. The framework described is shown in Figure 2 and consists of three main steps which are discussed below in detail.

Critical Variables Identification:

Scan the software repository to identify all the critical variables present in the source code. Critical variables are those which interact with the external world by accepting user input, and also which are part of critical operations that involve query executions.

Path Identification Function:

By combining data-flow and basis-path testing, legal execution paths of the application are obtained. Data-flow testing of the critical variables identifies all the legal sub-paths that can be taken by critical variables during execution. Basis-path testing is performed to identify the minimum number of legal execution paths of the application. Since basis-path testing leads to reduced number of monitorable paths, the complexity of our proposed technique in terms of integrating monitors across multiple paths also reduces. The path identification function builds the set of critical paths to be monitored in the application.

Let $C = \{C^1, C^2 \ldots\ldots, C^m\}$ be a set of *m* critical variables identified during critical variable identification phase.

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

Let $P = \{P^1, P^2 \ldots\ldots, P^k\}$ be a set of k legal paths identified using basis-path testing and CP is a set of paths we intend to monitor.

CP is identified using the pseudo code shown below:

$CP = \{ \}$
for every $P^j \in P$ *and*
  for every $P_C^i \in P_C$
    if $(P^j \cap P_C^i == P_C^i)$
      $CP = CP \; U \{ P^j \}$
where, $i \in [0, m]$ *and* $j \in [0, k]$

We thus identify all the critical paths of the application to be monitored.

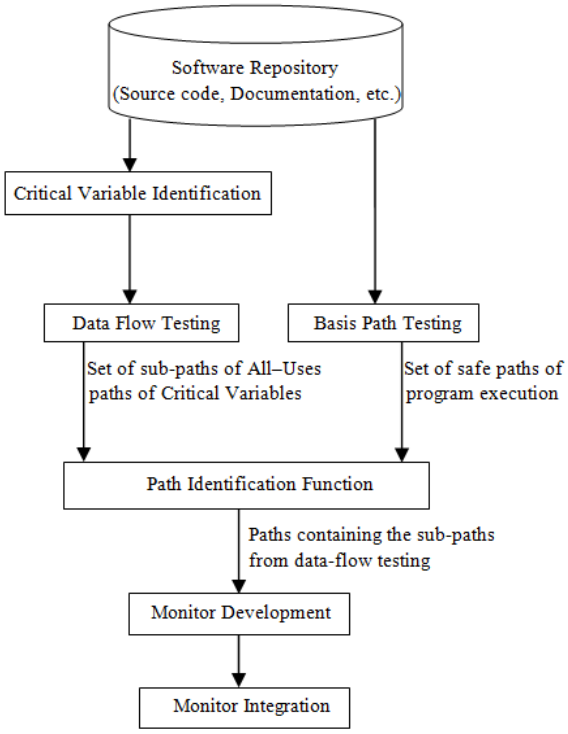

Figure 2. Runtime monitoring framework for tautology based SQLIAs.

Monitor Development and Integration:

In this phase, we develop a monitor for the identified critical paths using AspectJ [12]. The developed monitor is then integrated with the respective module of the application for monitoring the critical paths. Henceforth, on every query execution, the runtime monitor

tracks the identified critical variables by monitoring their execution path. When a critical variable follows an invalid path, the runtime monitor immediately detects the abnormal behavior of the application due to the critical variable and notifies the administrator.

Thus, using the above discussed phases in our proposed framework, we develop a runtime monitoring technique to handle tautology based SQLIAs that uses the knowledge gained from pre-deployment testing of web application to develop runtime monitors.

3 IMPLEMENTATION

To evaluate our approach, we developed a framework called Runtime Monitoring Framework to handle tautology based SQLIAs in Java based web applications. We chose to target Java because it is a commonly used language for developing web applications.

Figure 3 shows the high-level view of the Runtime Monitoring Framework. As the figure shows, the framework consists of the following modules: i) Critical Variables Identification Module ii) Critical Paths Identification Module iii) Runtime Monitor Development and Instrumentation Module.

Critical Variables Identification Module:

The Critical Variables Identification Module identifies all the critical variables, i.e. variables that are initialized with the input provided by external user and those that become a part of SQL query. Input to this module is a Java web application and it outputs the critical variables. In our present implementation, this is done manually and we intend to automate this process in our future implementation.

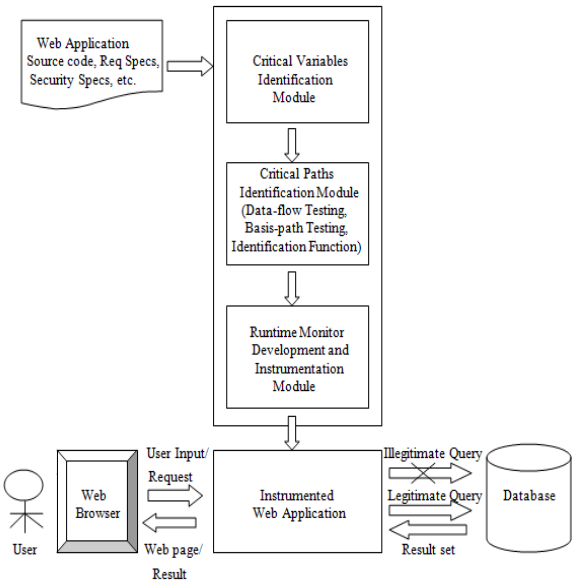

Figure 3: High Level View of Runtime Monitoring Framework.

Critical Paths Identification Module:

The Critical Paths Identification Module identifies the critical paths generated by data-flow and basis-path testing techniques. The module takes the identified critical variables as input and returns the paths that need to be monitored. Data-flow testing of the critical variables helps in identification of all the legal sub-paths that can be taken by critical variables during execution. Basis-path testing is performed to identify the minimum number of legal execution paths of the application. Since basis-path testing leads to reduced number of monitorable paths, the complexity of our proposed technique in terms of integrating monitors across multiple paths also reduces. The path identification function builds the set of critical paths to be monitored in the application to detect and prevent tautology based SQLIAs.

Runtime Monitors Development and Instrumentation Module:

This module develops the runtime monitor for the identified critical paths and instruments it to the appropriate part of the source code. AspectJ [12] is used to generate and integrate monitor into the application.

4 EVALUATION

In this section, we discuss the evaluation to assess effectiveness and efficiency of our proposed technique. Following are the research questions (RQ) for which we intend to find solutions.

RQ1: What percentage of tautology based SQLIAs can our proposed technique detect and prevent that would otherwise go undetected? (False Negative Rate)

RQ2: What percentage of legitimate accesses does our proposed technique identify as tautology based SQLIAs and prevents them from executing on the database? (False Positive Rate)

4.1 Experimental Setup

To be able to investigate the research questions, we developed an interactive web application called "Employee Information Retrieval Application" that accepts HTTP requests from a client, generate SQL queries, and issues them to the underlying database.

4.1.1 Subject

The subject application we developed for our experimentation purpose is an "Employee Information Retrieval Application." It accepts input from an external user through a web form, and uses the input to build queries to an underlying database, and retrieves the relevant information of the particular user. Front-end of the application is developed using HTML language, Java Servlet is used for processing the input received from the user and connecting to the back-end database for retrieving and displaying the information to the user. Also, MySQL database is used at the back-end to store the employee related information. The table "empinfo" consists of six fields namely: UserName, Password, SSN, Name, Age and Department.

When legitimate input i.e. username and password are provided by the user, the submitted credentials are then used to dynamically build the query as shown below:

*String query = "Select * FROM empinfo where username = '"athomas"' and password = '"andrew999"'";*

The query executes successfully and the application returns the relevant records to the user.

4.1.2 Application of Runtime Monitors to the subject.

In this section, we describe the results obtained when the runtime monitor developed using the proposed Runtime Monitoring Framework is instrumented into the web application discussed above.

When an illegitimate input such as ' OR 1 = 1 -- ' and ' ' is provided by an external user for username and password variables respectively, this causes a tautology based SQLIA on the

application. The submitted credentials are then used to dynamically build the query as shown below:

*String query = "Select * FROM empinfo where username = ' " OR 1 = 1 --" ' and password = ' " " ' ";*

The illegitimate input provided by the external user will cause the application to behave in an abnormal way by displaying all the records present in the database. The runtime monitor instrumented in the web application will detect this abnormal behavior of the application and halts the execution of the application. The monitor also notifies the administrator about the attack.

4.1.3 Discussion of Results

Table 1 summarizes the results obtained when a set of illegitimate tautology based inputs are provided to the instrumented web application described above. The Attack Detected column in the table will have "YES" value if the attack is detected successfully by our proposed technique, else it contains a "NO" value.

Table 1: Results obtained by the application of our framework to detect Illegitimate Queries

Illegitimate Inputs	Attack Detected
username: ' OR 1 = 1 -- ' password: ' '	**YES**
username: athomas password: ' OR 1 = 1 '	**YES**
username: ' '111' OR true# ' password: ' '	**YES**
username: mfranklin password: ' 'aaa' OR '1=1' '	**YES**
username: ' '111' OR 1=1 -- ' password: ' '	**YES**
username: athomas password: ' OR username between 'A' and 'Z	**YES**
username: mfranklin password: ' 'admin' OR '1>4' '	**YES**
username: mphelps password: ' 'admin' OR '4>1' '	**YES**
username: ' 'username' OR 1=1 -- ' password: ' '	**YES**
username: athomas password: ' 'password' OR '1=1' '	**YES**
username: ' 'admin' OR 1<2 -- ' password: ' '	**YES**
username: ' 'admin' OR 7>6 -- ' password: ' '	**YES**

Table 2 summarizes the results obtained when a set of legitimate inputs are applied to the instrumented web application. The Query Successful column will have a "YES" value in case of successful query execution, else a "NO" if the legitimate query is falsely detected as an attack.

Table 2: Results obtained by the application of our framework to detect Legitimate Queries

Legitimate Inputs	Query Successful
username: mdavid password: **********	**YES**
username: rrandy password: **********	**YES**
username: mfranklin password:**********	**YES**
username: JSmith765 password:**********	**YES**
username: Anderson9John password: **********	**YES**
username: LAdams password: *********	**YES**
username: sparker password: **********	**YES**
username: LauraAdams password: **********	**YES**
username: SGreenSFO password: **********	**YES**
username: parker765 password: **********	**YES**

We used 12 illegitimate tautology based inputs as shown in Table 1 and 10 legitimate inputs as shown in Table 2 for our evaluation. The results of our study

clearly demonstrate the success of our Runtime Monitoring Technique to handle tautology based SQLIAs. Our proposed technique was able to successfully allow all the legitimate queries to be executed on the application and detected all the tautology based SQLIAs i.e. both false positives and false negatives were handled effectively.

Though we have performed our experimentation on a simple target web application and for a small number of inputs, the preliminary results obtained are encouraging; because we have used all realistic tautology based attacks as illegitimate inputs for the instrumented application taken as subject. However, before drawing definitive conclusions, still more extensive experimentation is needed.

5 RELATED WORK

Over the past decade, a lot of work has been accomplished by the research community in providing new techniques to detect and prevent SQLIAs. In this section, we discuss state-of-the-art in SQLIA detection and prevention techniques and classify them into two categories namely: (i) Pre-deployment Testing Techniques and (ii) Post-deployment Testing Techniques.

5.1 Pre-deployment Testing Techniques

Pre-deployment techniques consist of methodologies which are used earlier in the Software Development Life Cycle i.e. before the software has been deployed in the real world to detect SQLIAs in web applications. Techniques discussed in this section also come under the category of static analysis using which the applications are tested for possible SQLIAs without executing the application.

Huang et al. in [13], proposed a tool WebSSARI which uses information flow analysis for detecting input validation related errors. It uses static analysis to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application to satisfy these preconditions. It works based on sanitized input that has passed through a predefined set of filters.

Wasserman et al. in [14], proposed a static analysis framework that operates directly on the source code of the application to prevent tautology attack. Static analysis is used to obtain a set of SQL queries that a program may generate as a finite state automaton. The framework then applies an algorithm on the generated automaton to check whether there is a tautology and the existence of a tautology indicates the presence of a potential vulnerability. The important limitation of Tautology Checker is that, it can detect only tautology based SQLIAs but cannot detect other types of SQLIAs.

Gould et al. in [15], describe about JDBC Checker, a sound static analysis tool to verify the correctness of dynamically generated query strings. JDBC Checker can detect SQL injection vulnerabilities caused by improper type checking of the user inputs. This technique would not catch more general forms of SQLIAs, but can be used to prevent attacks that take advantage of type mismatches in a dynamically-

generated query string. The root cause of SQL injection vulnerabilities in code, which is improper type checking of input can be detected by a JDBC-Checker. General forms of SQLIAs cannot be caught by this technique because most of these attacks consist of syntactically and type-correct queries.

Livshits et al. in [16], propose a tool based static analysis technique to detect SQL injection vulnerabilities in web applications. User-provided specifications of vulnerability pattern in PQL language are applied to Java byte code and all vulnerabilities matching a specification are found automatically in the statically analyzed code. With static analysis all potential security violations can be found without executing the application.

Fu et al. in [17], proposed SAFELI a static analysis tool which can automatically generate test cases exploiting SQL injection vulnerabilities in ASP.NET web applications. SAFELI instruments the bytecode of Java Web applications and utilizes symbolic execution to statically inspect security vulnerabilities. Whenever a hotspot which submits SQL query is encountered, a hybrid string equation is constructed to find out the initial values of Web controls which might be used to apply SQLIAs. Once the equation is successfully solved by a hybrid string solver, the solution of the equation is used to construct a test case which is replayed by an automated GUI testing tool. SAFELI can analyze the source code and will be able to identify delicate vulnerabilities that cannot be discovered by black-box vulnerability scanners. The main drawback of this technique is that, this approach can discover the SQLIAs only on Microsoft based products.

Mui et al. in [18], propose ASSIST to protect Java based web applications against SQLIAs. A combination of static analysis and program transformation is used by ASSIST to automatically identify locations of SQL injection vulnerabilities in code and instrument them with calls to sanitized functions. The automated technique will help developers to eliminate the tedious process of performing manual inspection and sanitization of code.

All the above mentioned techniques are used to detect SQLIAs in web application before they are deployed in the real world; these techniques use static analysis i.e. they do not execute the application to detect the vulnerabilities instead perform code check to verify for any possibility of attack. But, in reality a lot of SQLIAs occur once the software is deployed in the real world. In this perspective, our proposed framework is mainly focused on developing software runtime monitor that uses runtime monitoring technique to detect SQLIAs based on the behavior of the web application during its post-deployment.

5.2 Post-deployment Testing Techniques

Post-deployment techniques consist of dynamic analysis technique which can be used to detect SQLIAs in web applications after it has been deployed in the real world. In this section, we discuss about the existing techniques that come under the category of post-deployment techniques and compare them with our proposed approach.

Buehrer et al. in [19], present 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.

Halfond et al. in [7, 20, and 21], propose 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 which 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.

Su et al. in [22], proposed SQL-Check which is a runtime checking system. The technique used in SQL check will first track the user input substring in the program and syntactically track those substrings using a syntactic policy. This will specify all the permitted syntactic forms. This process forms an annotated query also called an augmented query. A parser is then used by SQL Check to parse the augmented query and to find whether the query is legitimate or not. If the query parses successfully, then the query is supposed to have met the syntactic constraints and is considered as legitimate. But, if the query has not successfully passed by the parser then it is considered to be a command injection attack query. This approach uses a secret key to discover user inputs in the SQL queries. Thus, the security of the approach relies on attackers not being able to discover the key. Also, this approach requires the application developer to either 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.

Bisht et al. in [23], 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. CANDID retrofits web applications written in Java through a program transformation and its natural and simple approach turns out to be very powerful for detection of SQLIA.

Combined static and dynamic analysis approaches as discussed above in [7, 19, 20, 21, 22 and 23] use static analysis technique to identify the intended structure of SQL queries in the absence of user inputs by analyzing the source code and constructing the syntactic models like parse trees. The proposed approaches then use dynamic analysis and detect SQLIA at runtime if the

syntactic structure of the dynamically generated query which includes user inputs deviates from the statically generated syntactic models. In our proposed approach pre-deployment testing techniques such as data-flow and basis-path are used to first find the valid/legal behaviors of the application in the presence of the user input. Runtime monitoring of the application is then performed with the developed monitors to see if the execution of the application deviates from the specified valid/legal path. Any deviation observed by the monitor is identified as the possible exploitation of SQLIA vulnerability and immediately notified to the administrator.

Halfond et al. in [2], 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 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.

Boyd et al. in [24], 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 because, the user input inserted into the randomized query will be classified as a set of non-keywords resulting in an invalid expression. SQLrand requires the developers to randomize SQL queries present in the application by appending a random integer value, so its security relies on attackers not being able to discover the integer value. In my proposed method SQL queries will be written using standard keywords and the monitors will be developed and instrumented into the source code automatically. Also, the need for the deployment of proxy is eliminated.

Pietraszek et al. in [25], introduced CSSE, a method to detect and prevent injection attacks. CSSE works by automatic marking of all user-originated data with meta-data about its origin and ensuring that this metadata is preserved and updated when operations are performed on the data. The metadata enables a CSSE-enabled platform to automatically carry out the necessary checks at a very late stage and it is able to independently determine and execute the appropriate checks on the data it previously marked unsafe. CSSE is transparent to the application developer, as the necessary checks are enforced at

the platform level and neither modification nor analysis of the application is required.

Huang et al. in [26], proposed WAVES a blackbox 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 monitors the application's response to the attacks by utilizing machine learning to improve its attack methodology. WAVES is better than traditional penetration testing, because it improves the attack methodology, by using machine learning approaches to guide its testing.

Valeur et al. in [27], 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 training phase, profiles are learned automatically by analyzing a number of sample database accesses. During detection phase, anomalous queries that lead to SQLIA are identified. IDS detect attacks successfully but, the overall IDS quality depends on the quality of the training set and they generate a large number of false alarms.

Cova et al. in [28], present Swaddler, an approach for the detection of attacks against web applications based on the analysis of the internal application state. Swaddler analyzes the internal state of a web application and learns the relationships between the application's critical execution points and the application's internal state. The approach is based on a detailed characterization of the internal state of a web application, by means of a number of anomaly models. The internal state of the application is monitored during the learning phase. During this phase the approach derives the profiles that describe the normal values for the application's state variables in critical points of the application's components. Then, during the detection phase, the application's execution is monitored to identify anomalous state.

Most of the post-deployment techniques discussed above generate a meta-model of possible attack queries during the learning phase of software execution. The queries then formed every time due to the input provided by an external user is compared with the generated meta-model and appropriate decisions are made. Since these techniques are mainly dependent on the accuracy of the learning phase, it is possible that few of the SQLIAs may go unnoticed causing threat to the database. In order to overcome this, in our approach we monitor the legitimate behavior of the application during its execution to handle SQLIAs.

6 CONCLUSION

In this paper, we introduced a new technique to handle tautology based SQLIAs. We also propose a framework called Runtime Monitoring Framework used by our technique for development of runtime monitors, which perform runtime monitoring of a web application during its post-deployment to detect and prevent tautology based SQLIAs. Thus, using our framework, we ensure that the

quality and security of the application is achieved not only during its pre-deployment, but also during its post-deployment phase, and any possible exploitation of vulnerability by an external attacker is detected and prevented. We also presented the evaluation of our proposed technique. The results obtained clearly indicate that our technique was successfully able to handle all of the tautology based SQLIAs and allowed legitimate inputs to access the database.

We further intend to automate the entire process of using the proposed framework to develop the runtime monitors and also extend the framework to detect and prevent all other types of SQLIAs.

7 REFERENCES

[1] OWASP – Open Web Application Secuirty Project. Top ten most web application vulnerabilities. http: //ww.owasp.org/index.php/OWASP_TOP_Ten_ Project, April 2010.

[2] W. G. J. Halfond, A. Orso and P. Manolios, "Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks", Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2006.

[3] W. G. J. Halfond, J. Viegas, and A.Orso, "A Classification of SQL - Injection Attacks and Countermeasures", Proceedings of the IEEE International Symposium on Secure Software Engineering, 2006.

[4] A. Tajpour and M. Massrum, "Comparison of SQL Injection Detection and Prevention Techniques", In 2nd International Conference on Education Technology and Computer, 2010.

[5] G. Erdogan, "Security Testing of Web Based Applications", Norwegian University of Science and Technology (NTNU), 2009.

[6] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Vishwanathan, "Computational Analysis of Runtime Monitoring - Fundamentals of Java-Mac", RV'02 Runtime Verification 2002, Volume: 70, Issue: 4, Dec 2002.

[7] W. G. J. Halfond and A. Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL Injection Attacks", Proceedings of 3rd International Workshop on Dynamic Analysis, 2005.

[8] Software Secuirty Testing, Software Assurance Pocket Guide Series: Development, Volume III, Version 1.0, May 21, 2012.

[9] Ramya Dharam, Sajjan. G. Shiva, "A Framework for Development of Runtime Monitors", International Conference on Computer and Information Sciences (ICCIS), Kuala Lumpur, Malaysia, June 2012.

[10] K. Saleh, A. S. Boujarwah, J. Al-Dallal, "Anomaly Detection in Concurrent Java Programs Using Dynamic Data Flow Analysis", Information and Software Technology, Volume: 43, Issue: 15, December 2001.

[11] Mohd. Ehmer Khan, "Different Approaches to White Box Testing for finding Errors", International Journal of Software Engineering and Its Applications, Vol. 5, N0. 3, July 2011.

[12] AspectJ Cookbook, Russ Miles, December 27, 2004.

[13] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee and S. Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection", Proceedings of 13th International Conference on World Wide Web, 2004.

[14] G. Wassermann and Z. Su, "An Analysis Framework for Security in Web Applications", Proceedings of the FSE Workshop on Specification and Verification of Component Based Systems, 2004.

[15] C. Gould, Z. Su and P. Devanbu, "JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications", Proceedings of the 26th International Conference on Software Engineering, 2004.

[16] V. B. Livshits and M. S. Lam, "Finding Security Errors in Java Programs with Static Analysis", Proceedings of the 14th Usenix Security Symposium, 2005.

[17] X. Fu and K. Qian, "SAFELI – SQL Injection Scanner Using Symbolic Execution", Proceedings of 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, 2008.

[18] R. Mui and P. Frankl, "Preventing SQL Injection through Automatic Query Sanitization with ASSIST", Fourth International Workshop on Testing, Analysis and Verification of Web Software, 2010.

[19] G. T. Buehrer, B. W. Weide and P. A. G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks", International Workshop on Software Engineering and Middleware, 2005.

[20] W. G. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks", Proceedings of the IEEE and ACM International Conference on Automated Software Engineering, Nov 2005.

[21] W. G. Halfond and A. Orso, "Preventing SQL Injection Attacks Using AMNESIA", Proceedings of 28th International Conference on Software Engineering, 2006.

[22] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in web Applications", The 33rd Annual Symposium on Principles of Programming Languages, 2006.

[23] P. Bisht and P. Madhusudan, "CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks", Proceedings of the 14th ACM Conference on Computer and Communications Security, 2007.

[24] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL Injection Attacks", Proceedings of the 2nd Applied Cryptography and Network Security Conference, June 2004.

[25] T. Pietraszek and C. V. Berghe, "Defending Against Injection Attacks Through Context-Sensitive String Evaluation", Proceedings of Recent Advances in Intrusion Detection, 2005.

[26] Y.W. Huang, S. K. Huang, T. P. Lin & C. H. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring", Proceedings of the 12th International Conference on World Wide Web, 2003.

[27] F. Valeur, D. Mutz, and G. Vigna, "A Learning Based Approach to the Detection of SQL Attacks", Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment, 2005.

[28] M. Cova, D. Balzarotti, "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications", Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection, 2007.