Runtime Monitors to Detect and Prevent Union Query based SQL Injection Attacks

Ramya Dharam[1] and Sajjan. G. Shiva[2]
[1]*Department of Computer Science, University of Memphis, Memphis, TN, USA*
[2]*Department of Computer Science, University of Memphis, Memphis, TN, USA*

Abstract

Web applications are increasingly used in recent years to provide online services such as banking, shopping, social networking, etc. These applications operate with sensitive user information and hence there is a high need for assuring their confidentiality, integrity, and availability. Existing pre-deployment testing techniques, tools, and methodologies do not assure complete analysis, execution and testing of all possible behaviors of the software. This causes the software to sometimes behave differently than what it was designed for during its post-deployment. Such a deviation in the system's behavior, also termed as "Software Anomaly," is mostly due to external attacks such as Path Traversal Attacks, SQL Injection Attacks, etc., that in turn affect confidential user information stored in the application. In this paper, we present and evaluate a framework called Runtime Monitoring Framework to handle union query based SQL Injection Attacks.

Keywords: Runtime Monitors, Union Queries, SQL Injection Attacks, Data-flow Testing, Basis-path Testing.

1. Introduction

Database-driven web applications have been widely used by organizations to provide a broad range of services to their users. These applications contain database at their back-end to store confidential user data, like financial, medical, and personal information records, etc., that makes web applications an ideal target for attacks. SQL Injection Attacks (SQLIAs) have been identified as one of the major security threats to web applications [1]. They give attackers unauthorized access to the underlying database and also the rights to retrieve, modify and delete valuable user information stored in the database resulting in security violations, identity theft, etc.

SQLIAs occur when data provided by an external user is included directly in a SQL query and is not properly validated. Inadequate input validation within an application has been identified as one of the major cause for SQLIAs. Implementing input validation routines can serve as a first level of defense against SQLIAs, but they cannot defend against sophisticated attack techniques that inject malicious inputs into SQL queries [2, 3]. A variety of programming practice guidelines and web application security testing tools and scanners have also been proposed by the research community to detect and prevent SQLIAs. Tools such as firewalls and Intrusion Detection Systems (IDSs) are ineffective against SQLIAs, because ports which are open in firewalls for regular web traffic in the application level are used to perform SQLIAs [4]. In spite of implementing the described detective and preventive techniques, attackers are still able to successfully perform SQLIAs on web applications and gain unauthorized access to the confidential user information.

In this paper, we introduce a framework called Runtime Monitoring Framework to develop runtime monitors. These monitors perform post-deployment monitoring of the application to detect and prevent union query based SQLIAs. The framework introduced in this paper is an extension of our framework proposed in [5] to handle tautology based SQLIAs. Initially, the framework uses two pre-deployment testing techniques i.e. basis-path and data-flow testing techniques to help in the development of runtime monitors for all the identified legal/valid execution paths. Then, monitors are integrated into the respective module of the application to perform runtime monitoring of the application during its pos-deployment for the identified legal/valid execution paths. Any deviation in the

behavior of the application will be identified by the runtime monitor as a possible exploitation of union query based SQLIAs and halts the execution of the application. The runtime monitor also notifies the administrator about the attack. In this paper, we also present the preliminary results obtained when the runtime monitor developed using the framework is instrumented into a target web application to detect and prevent union query based SQLIAs.

The paper is organized as follows. In Section 2, we discuss our research strategy and methodology. Evaluation and results obtained are discussed in Section 3. In Section 4, we discuss about related work and conclude in Section 5.

2. Research Strategy and Methodology

In this section, we describe about statement injection based SQLIAs and more specifically union query based SQLIAs. We also discuss about the research strategy and methodology used for the development of the framework and finally overview of our framework is discussed.

2.1 Modeling Union Query based SQL Injection Attacks

A web application structure is a three-tiered architecture which consists of a web browser, an application server, and a back-end database server. A web application with such architecture will initially receive input from an external user, and then dispatch the queries to the underlying database for execution. Finally, the application will then retrieve and present data to the user based on the input provided.

Serious security problems can arise in such application if the user 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. Statement Injection Attacks is a type of SQLIAs, using which attacker inserts additional statements consisting of union operator into the original SQL statement via user input. The union operator present in SQL is used to join multiple tables together. If an attacker inserts code containing the union operator, then the attacker is trying to return more information than the query intended. Therefore union query based SQLIAs aims to compromise data confidentiality.

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 " ' union select * from employeesal -- " and " ' ' " instead of a legitimate username and password into their respective fields the query string becomes as follows:

*SELECT * FROM employeeinfo WHERE name = ' ' union select * from employeesal -- ' AND password =' ' ' ';*

Any website that uses this code would be vulnerable to union query based SQLIAs. The character "--" indicates the beginning of a comment, and everything following the comment is ignored. Therefore, the query now becomes the union of two select queries. The first select query returns a null set because there are no matching records in employeeinfo table. The second select query returns all the records from the table employeesal. When the above query is executed the user will bypass the authentication logic and the union query will retrieve all records from the table employeesal. Thus, the attacker successfully gains unauthorized access to the sensitive and valuable information about all the users stored in a different table.

2.2 Runtime Monitoring Framework for Union Query based SQL Injection Attacks

Our framework uses the information gathered from pre-deployment testing of web application, to help in development of runtime monitor to detect and prevent union query based SQLIAs. The framework initially uses a software repository which consists of a collection of documents related to requirements, security specifications, and application source code, etc., to find the critical variables. 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 [6] is an effective approach to detect improper use of data and can be performed either statically or dynamically.

Basis-path testing [7] is a white box testing technique that identifies the minimal set of all legal execution paths 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 1 and consists of three modules which are discussed below in detail.

Critical Variables Identification (CVI): variables which accept external user input, and also are part of critical operations that involve query executions are termed as *Critical Variables*. The CVI module accepts a Java based web application source code as an input and outputs all the Critical Variables present in the application. In our present implementation, the identification process is done manually and we intend to automate this process in our future implementation.

Path Identification Function (PIF): The Path taken by the critical variables during their execution is identified by PIF module. The module takes the critical variables - identified during CVI module in the previous step, as input and returns the paths that need to be monitored. The module uses a combination of data-flow and basis-path testing techniques to generate the paths. 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.

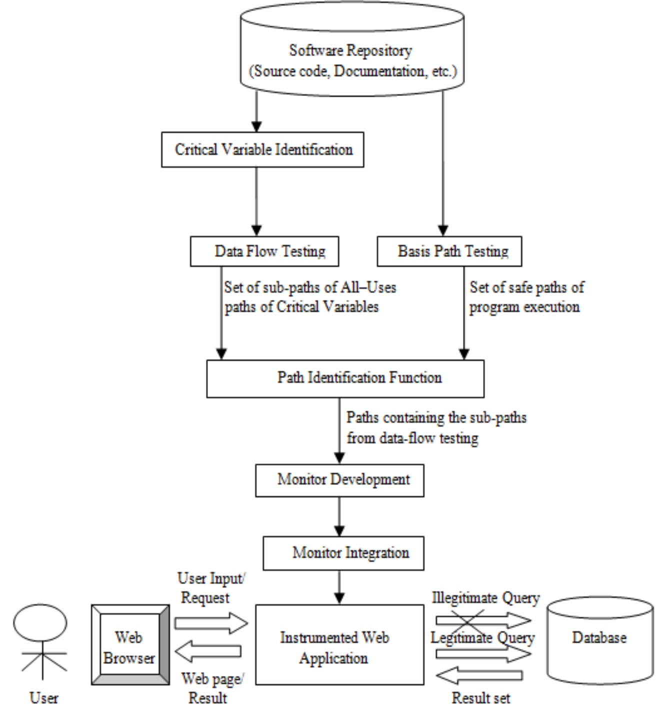

Figure 1: Overview of Runtime Monitoring Framework

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

Let $P_C = \{\{ P_C^1 \} \cup \{ P_C^2 \} \cup \ldots\ldots, \{ P_C^m\}\}$ 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 ϵ P and
    for every P_C^i ϵ P_C
        if (P^j ∩ 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.

Monitor Development and Integration (MDI): This module develops the runtime monitor for identified critical paths and instruments it into the appropriate part of the source code. AspectJ [8] is used to generate and integrate monitor into the application. 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.

3. Evaluation and Results

In this section, we discuss the results obtained when the runtime monitor developed using our proposed framework is instrumented into the subject web application discussed below.

For the experimentation purpose, we developed an interactive web application called "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 application consists of two tables named empinfo and empsal. The table "empinfo" consists of six fields namely: UserName, Password, SSN, Name, Age, and Department. Another table "empsal" consists of six fields namely: Name, Department, Title, Sal, Position_Id, and EmailId.

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 = '"aanthony"' and password = '"andrewSFO"'";*

When an illegitimate input such as ' union select * from empsal -- and ' ' is provided by an external user for username and password variables respectively, this causes a type of statement injection based SQLIAs (more specifically a union query based SQLIAs) on the subject application. By inserting "union" query in the input field the attacker is trying to gain access to all the records present in another table named "empsal". The submitted credentials are used to dynamically build the query as shown below:

*String query = "Select * FROM empinfo where username = ' " ' union select * from empsal -- " ' and password = ' " " '";*

The two dashes at the end of the union query comments out the rest of the query. The first Select query returns a null set because there is no matching record in the table empinfo. The second query will try to return all the data from the empsal table. Thus, the illegitimate union query based SQLIAs provided by the external user will cause the application to behave in an abnormal way by displaying all the records present in another table named empsal. The runtime monitor instrumented in the subject web application will detect this abnormal behavior of the application trying to display all the records present in another table and halts the execution of the application. The monitor also notifies the administrator about the possible union query based SQLIAs.

The experimentation performed clearly demonstrates the success of the developed runtime monitor to handle union query based SQLIAs on the subject application. The developed monitor successfully allowed all the legitimate queries to be executed on the application and detected all the union query based SQLIAs i.e. both false positives and false negatives were handled effectively.

Though a simple target web application along with small number of inputs has been used to perform our experimentation, the preliminary results obtained are encouraging. However, more extensive experimentation is needed before drawing definitive conclusions.

4. Related Work

The state-of-the-art in SQLIAs detection and prevention techniques is discussed in this section. The techniques are classified into two categories namely: (i) Pre-deployment Techniques and (ii) Post-deployment Techniques.

4.1 Pre-deployment 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.

Wasserman et al. [9] proposed a static analysis framework that operates directly on the source code of the application to prevent tautology based SQLIAs. 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 and prevent only tautology based SQLIAs, which is only one of the many kinds of SQLIAs that our technique addresses.

Livshits et al. [10] use static analysis techniques to detect SQL injection vulnerabilities in web applications. User-provided specifications of vulnerability pattern in PQL language are used to find all vulnerabilities matching the specification. The primary limitation of this approach is that it can only detect know and specified vulnerability patters of SQLIAs and cannot detect SQL injection attack patterns that are not known beforehand.

Fu et al. [11] proposed SAFELI a static analysis tool which can automatically generate test cases exploiting SQL injection vulnerabilities in ASP.NET web applications. SAFELI analyzes the source code of the applications and identifies SQL injection vulnerabilities. The main drawback of this technique is that, this approach can discover the SQLIAs only on Microsoft based products.

Mui et al. [12] proposed ASSIST to protect Java based web applications against SQLIAs. A combination of static analysis and program transformation is used by ASSIST which first uses static analysis to find host variables and automatically sanitizes them by instrumenting them with calls to sanitized functions.

All the above mentioned techniques are used to detect SQLIAs in web application during its pre-deployment; these are static techniques which employ the use of static code analysis to identify the source of injection vulnerabilities in code or occurrences of attacks. A lot of SQLIAs occur once the software is deployed in the real world and in this perspective, our proposed technique uses dynamic analysis and in particular runtime monitoring technique to detect and prevent SQLIAs based on the behavior of the web application during its post-deployment.

4.2 Post-deployment Techniques

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

Buehrer et al. [13] 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. [14] 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.

The approaches discussed above 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. 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, such as data-flow and basis-path, are used to find the valid/legal behaviors of the

application in the presence of user input; during runtime, the developed monitors perform the runtime monitoring to observe if the execution of the application deviates from the specified valid/legal path.

5. Conclusion

In this paper, we presented a Runtime Monitoring Framework for development of runtime monitors, which perform runtime monitoring of a web application during its post-deployment to detect and prevent union query 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. The results obtained indicate that the runtime monitor was successfully able to handle all the union query based SQLIAs on the subject web application and allowed legitimate inputs to access the database. We further intend to automate the entire process of the development of monitors and extend the framework to detect and prevent all other types of SQLIAs.

6. 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”, In SIGSOFT’06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2006.

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

[4] 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.

[5] R. Dharam and S. Shiva, “Runtime Monitors for Tautology based SQL Injection Attacks”, International Conference on Cyber Security, Cyber Warfare and Digital Forensics, Kaula Lumpur, Malaysia, June 2012.

[6] 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.

[7] Mohd. Ehmer Khan, “Different Approaches to White Box Testing Technique for finding Errors” , International Journal of Software Engineering and Its Applications, Vol. 5, No. 3, July 2011.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.