ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering
(An ISO 3297: 2007 Certified Organization) **Vol.2, Special Issue 1, March 2014**

Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)

Organized by

Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

An Effective Approach for Detecting and Preventing Sqlinjection Attacks

M. Roslinmary[1], S. Sivasakthi[2], A. Shenbaga Bharatha Priya[3]

PG scholar, Department of IT, Dr. Sivanthi Aditanar College of Engineering, Tiruchendur, Tamilnadu, India[1, 2, 3]

ABSTRACT*:* Many software systems have evolved to include a Web-based component that makes them available to the public via the Internet and can expose them to a variety of Web-based attacks mainly SQL-Injection attack. SQL-Injection attack will give attackers unrestricted access to the database. SQL-Injection Preventer prevents various set of database attacks and also security problems related to input validation. This is a highly automated approach for protecting the web applications against SQL-Injection and it has more practical advantages than that of existing techniques. For example, usage of defensive coding practices in most of existing systems. This technique is precise and efficient and has minimum deployment requirements. The methodologies used behind this approach are positive tainting and flexible syntax aware evaluation. Positive tainting marks and tracks certain data in a program at run time. This project is implemented on credit card based application that prevents unauthorized access to the database by attackers and also provide proper relevant transaction needed by the user.

KEYWORDS– Database management system, SQL, SQL Injection Attacks.

I. INTRODUCTION

SQL injection is a technique used to take advantage of non-validated input vulnerabilities to pass SQL commands through a Web application for execution by a backend database. Attackers take advantage of the fact that programmers often chain together SQL commands with user-provided parameters, and can therefore embed SQL commands inside these parameters. The result is that the attacker can execute arbitrary SQL queries and or commands on the backend database server through the Web application. Databases are fundamental components of Web applications. Databases enable Web applications to store data, preferences and content elements. Using SQL, Web applications interact with databases to dynamically build customized data views for each user. A common example is a Web application that manages products. In one of the Web application's dynamic pages (such as ASP), users are able to enter a product identifier and view the product name and description. The approach used here to prevent the SQL Injection is Positive tainting and Syntax aware evaluation.

In this project, we propose a highly automated approach for detecting and preventing SQL Injection Attacks (SQLIAs). The approaches works by

(1) Identifying "trusted" strings in an application and
(2) Allowing only these trusted strings to be used to create sensitive parts of SQL query strings, such as keywords and operators.

The general mechanism that we use to implement this approach is based on the idea of dynamic taint propagation-input is marked and tracked during usage by the application and prevented from being used in ways that could cause harm to the system. While numerous techniques have been proposed in the literature for addressing the SQLIA problem, our use of dynamic positive tainting has several important advantages. Positive tainting is purely dynamic; it is both precise and efficient. Many previously proposed techniques require extensive involvement by the programmer. The conceptual advantages of our approach are in the use of positive Tainting and a flexible syntax-aware evaluation. In the proposed system, the SQL Injection prevents the hacker's attack with the use of this application in 10 ways. A new automated technique for preventing SQLIAs based on the concept of positive tainting and on flexible syntax-aware evaluation.

SQLIAs are a class of code injection attacks that take advantage of the lack of validation of user input. These attacks occur when developers combine hard-coded strings with user-provided input to create dynamic queries. Intuitively, if user input is not properly validated attackers may be to change the developers intended sql commands by inserting new SQL keywords or operators through specially crafted input strings. Interested reader scan refer to the work of Su and Wassermann for the formal definition of SQLIAs. SQLIAs leverage a wide range of mechanisms and input channels to inject malicious commands into a vulnerable application, Before providing a detailed discussion of these various mechanisms, we introduce an example application that contains a simple SQL Injection vulnerability and show how an attacker can leverage that vulnerability,

Our approach against SQLIAs is based on dynamic tainting, which has previously been used to address security problems related to input validation. Traditional dynamic tainting approaches mark certain untrusted data(typical user input)as tainted, track the flow of tainted data at runtime and prevent this data from being used in potentially harmful ways. Our approach makes several conceptual and practical improvements over traditional dynamic tainting approaches by taking advantages of the characteristics of SQLIAs and web applications,First,unlike existing dynamic tainting techniques, our approaches based on the concept of positive tainting, that is, the identification and marking of trusted, instead of untrusted data.Second,our approach performs accurate and efficient taint propagation by precisely tracking trust markings at the character level.Third,it performs syntax-aware evalution of queries whose nonliteral parts contain one or more characters without trust markings.Finally our approach has minimal deployment requirements which makes it both practical and portable.

II. PROBLEM DESCRIPTION

The various attacks that are used to attack the database of web applications are explained as below,

2.1 Special Symbol based Attack

User Name: abc'- -
Pass Word : 0

In SQL, "—" is the comment operator and everything after it is ignored. When performing this query, the database simply searches for an entry where login is equal to abc and returns that database record.

2.2 Tautologies

User Name: 'or 1=1- -
Pass Word : 0

This query will return account information for all of the users in the database. The attacker could log in as the first user in the 'users' table.

2.3 Like Keyword Queries

User Name: ' or user_info.LoginID like 'n%'
Pass Word : 0

The outcome of this attack is that the database returns where name like n.

2.4 Piggybacked Queries

User Name: abc
Pass Word : 0;drop table users

The database treats this query string as two queries separated by the query delimiter(";") and executes both. The second malicious query causes the database to drop the users in the database, which can delete all user information.

2.5 Shutdown Server

User Name: ';shutdown with nowait;- -
Pass Word : 0

The outcome of this injection is shutdown the server.

III. POSITIVE-TAINTING AND SYNTAX AWARE EVALUATION

In our approach contains three techniques by use of this technique we can find the injection data's and send the correct query to the sqlserver.

1. Positive-Tainting
2. Syntax aware
3. Character-level tainting.

Positive tainting focuses on the identification and marking of trusted data. In contrast, the standard "negative" tainting identifies and marks untrusted data. In the context of preventing SQLIAs , there are several reasons why positive tainting is more effective than negative tainting.

First, in web applications, trusted data sources can be more readily identified than untrusted data sources, therefore the use of positive tainting leads to increased automation. Second, and more importantly, the two approaches differ significantly with respect to incompleteness. When using positive tainting, failure to identify the set of trusted data sources can lead only to false positives, and such incompleteness tends to be easy to detect and correct during testing. In contrast, when using approaches based on negative tainting, failure to identify all untrusted data sources will leave the application vulnerable to attacks. In addition, such vulnerabilities are difficult to detect during testing, in part because untrusted data may come from sources such as user input, uploaded files, browser cookies, and local server variables. Failure to identify one of these sources as untrusted can result in SQLIAs in the field that may never be discovered. In contrast, incompleteness with positive tainting will result in false positives, which are undesirable, but whose presence can be detected immediately and which can be easily corrected.

The second conceptual advantage of our approach is due to our use of flexible syntax-aware evaluation, which provides developers with the mechanism to regulate the usage of strings based not only on their source, but also on the syntactical role they play in the final query string. This allows developers to use a wide range of external sources of input to build queries, while protecting them from possible attacks introduced via these sources.

3.1 Positive Tainting

Positive tainting consists of the identification and marking of data coming from trusted sources. This approach contrasts with the conventional dynamic tainting approach, negative tainting and show how using positive instead negative has significant implications for the effectiveness of the overall approach. One of the main problems with tainting-based approaches is tha correct identification of all relevant data tha should be marked. With negative tainting, incompleteness may generate false negatives, that is, it may leave an application vulnerable to attacks, and the problem may not be discovered until after an attack has actually occurred. With positive tainting, however, incompleteness in the identification of the data to be marked leads only to false positives that can be eliminated by the developer while keeping the system protected from attacks. In the context of SQL injection, this conceptual difference between positive and negative tainting is especially significant.

The characteristics of SQLIAs make the complete identification of all the sources of untrusted data problematic and, most importantly, the identification of trusted sources relatively straightforward. SQLIAs typically

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering
(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014

Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)
Organized by
Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

target web applications, which are highly modular, deployed in many different configurations, and interface with a wide range of external systems. In these applications there are often many different sources of external input to be considered. Enumerating all of these potential sources of untursted input is inherently error prone and unsafe. Case in point, developers initially assumed that only user input needed to be marked as tainted. However, subsequent exploits demonstrated that additional sources of external input, such as browser cookies and uploaded files, also needed to be marked.

Therefore, these input sources were added to the list of untrusted sources. Unfortunately, even this solution was not sufficient, as attackers soon realized they could use local server variables and the database itself as injection vectors. Even adding these additional sources is not guaranteed to provide a complete solution. If even a single of malicious input is left unidentified, data from that source would be implicitly trusted, which would leave an application vulnerable to attacks. The use of positive tainting addresses this problem. Identifying trusted, instead of untrusted data in a web application is in most cases straight forward because the set of trusted strings and trusted input sources that are used to build query strings are finite and typically small.

Our approach works by first accurately and automatically identify all hard-coded strings in the application and marking these strings as trusted. Hard coded strings are string which has been defined by the developer. For the common case of applications in which developers create SQL queries by combining hard-coded strings that contain SQL keywords and operators with user inputs that are used only as literals, this set is complete(i.e., it includes all and only the trusted data).E.g.: SELECT, FROM, UNION. These keywords are said to be trusted data.

Benefits of Positive tainting

- Increased safety: Incompleteness leads to easy-to-eliminate false positives.
- Normal in-house testing causes set of trusted data to converge to complete set.
- Increased automation: Trusted data readily identifiable in web application.

$\boxed{\texttt{SELECT}}\ \underline{\texttt{acct}}\ \boxed{\texttt{FROM}}\ \underline{\texttt{users}}\ \boxed{\texttt{WHERE}}\ \underline{\texttt{login}}\ \boxed{=}\ \boxed{'}\ \texttt{doe}\ \boxed{'}\ \boxed{\texttt{AND}}\ \underline{\texttt{pin}}\ \boxed{=}\ \texttt{123}$

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering
(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014

Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)
Organized by
Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

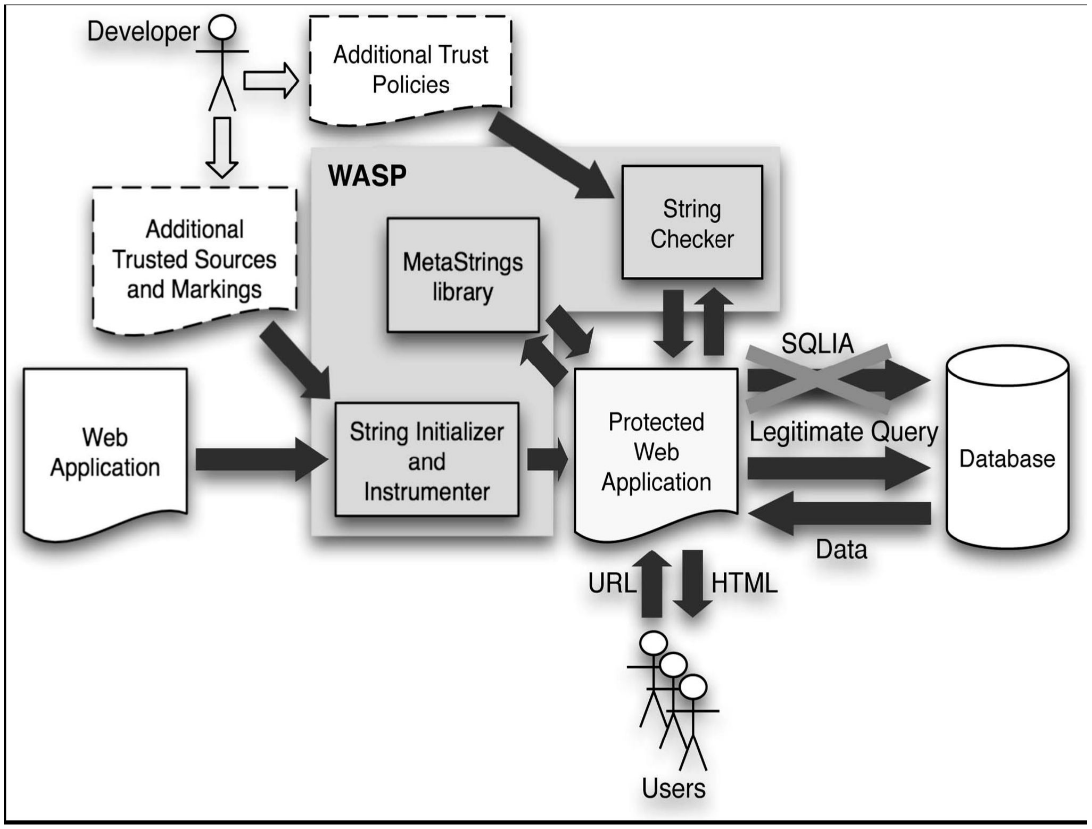
Developer
Additional Trust Policies
WASP
String Checker
MetaStrings library
Additional Trusted Sources and Markings
SQLIA
Web Application
String Initializer and Instrumenter
Protected Web Application
Legitimate Query
Database
Data
URL
HTML
Users

Figure.1: Diagram for SQLInjection Prevention System

3.2 Syntax-Aware Evaluation

Aside from ensuring that taint markings are correctly created and maintained during execution, our approach must be able to use the taint markings to distinguish legitimate from malicious queries. Simply forbidding the use of un trusted data in SQL commands is not available solution because it would flag any query that contains user input as an SQLIA, leading to many false positives. To address this shortcoming, researchers have introduced the concept of declassification, which permits the use of tainted input as long as it has been processed by a sanitizing function. (A sanitizing function is typically a filter that performs operations such as regular expression matching or substring replacement.) The idea of declassification is based on the assumption that sanitizing functions are able to eliminate or neutralize harmful parts of the input and make the data safe. However, in practice, there is no guarantee that the checks performed by a sanitizing function are adequate. Tainting approaches based on declassification could therefore generate false negatives if they mark as trusted supposedly sanitized data that is actually still harmful. Moreover, these approaches may also generate false positives in cases where un sanitized but perfectly legal input is used within a query.

Syntax-aware evaluation does not rely on any (potentially unsafe) assumptions about the effectiveness of sanitizing functions used by developers. It also allows for the use of un trusted input data in a SQL query as long as the use of such data does not cause an SQLIA. The key feature of syntax aware evaluation is that it considers the context in which trusted and un trusted data is used to make sure that all parts of a query other than string or numeric literals (for example, SQL keywords and operators) consist only of trusted characters. As long as un trusted data is confined to literals, we are guaranteed that no SQLIA can be performed. Conversely, if this property is not satisfied (for example, if a SQL operator contains characters that are not marked as trusted),we can assume that the operator has been injected by an attacker and identify the query as an attack.

Our technique performs syntax-aware evaluation of a query string immediately before the string is sent to the database to be executed. To evaluate the query string, the technique first uses a SQL parser to break the string into a sequence of tokens that correspond to SQL keywords, operators, and literals. The technique then iterates through the tokens and checks whether tokens (that is, substrings) other than literals contain only trusted data. If all such tokens pass this check, the query is considered safe and is allowed to execute. If an attack is detected, a developer specified action can be invoked.

This approach can also handle cases where developers use external query fragments to build SQL commands. In these cases, developers would specify which external Consider the malicious query, where the attacker submits "admin' – –" as the login and "0" as the pin. shows the sequence of tokens for the resulting query, together with the trust markings. Recall that "– –" is the SQL comment operator, so everything after this is identified by the parser as a literal. In this case, the Meta Checker would find that the last two tokens, ' and __ , contain un trusted characters. It would therefore identify the query as an SQLIA.

3.3 Character Level Tainting:

We track taint information at the character level rather than at the string level. We do this because, for building SQL queries, strings are constantly broken into substrings, manipulated, and combined. By associating taint information to single characters, our approach can precisely model the effect of these string operations. Another alternative would be to trace taint data at the bit level, which would allow us to account for situations where string data are manipulated as character values using bitwise operators.

However, operating at the bit level would make the approach considerably more expensive and complex to implement and deploy. Most importantly, our experience with Web applications shows that working at a finer level of granularity than a character would not yield any benefit in terms of effectiveness. Strings are typically manipulated using methods provided by string library classes and we have not encountered any case of query strings that are manipulated at the bit level .Accounting for string manipulations. To accurately maintain character-level taint information, we must identify all relevant string operations and account for their effect on the taint markings (that is, we must enforce complete mediation of all string operations).

Our approach achieves this goal by taking advantage of the encapsulation offered by object oriented languages, in particular by Java, in which all string manipulations are performed using a small set of classes and methods. Our approach extends all such classes and methods by adding functionality to update taint markings based on the methods' semantics.

IV. CONCLUSION

Our approach presented a novel highly automated approach for protecting web applications from SQLIAs. Our approach consists of identifying trusted data sources and marking data coming from these sources as trusted using dynamic tainting to track data to form the semantically other parts of queries such as SQL keywords and operators. Unlike previous approaches based on dynamic tainting, our technique is based on positive tainting which explicitly identifies trusted data in a program, this way; we eliminate the problem of false negatives that may result from the incomplete identification of all trusted data sources. False positives, although possible in some cases, can typically eliminate during testing, Our approach provides many advantages over the several existing techniques whose application requires customized and complex runtime environment. It is defined at the application level requires no modification of the runtime system and imposes a low execution overhead, We have evaluated our approach by developing a prototype tool, WASP and using the tool to protect 10 applications when subjected to a large and varied set of attacks and legitimate access.

REFERENCES

[1] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, "A Static Analysis Framework For Detecting SQL Injection Vulnerabilities", IEEE Dynamic SQL Transaction of computer software and application conference, 2007.

[2] William G.J. Halfond, Alessandro Orso,Panagiotis Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation", IEEE Transaction of Software Engineering Vol 34,Nol,Twentieth January/February 2008.

[3] Konstantinos Kemalis and Theodoros Tzouramanis, "Specification [18] Xin based approach on SQL Injection detection", ACM, 2008. *Building Secure Software: How to Avoid Security Problems the Right Way*, by John Viega and Gary McGraw. Addison-Wesley Professional, 2001.

[4] V. Benjamin Livshits and Monica S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis", ACM, 2005.

[5] Ashish kamra, Elisa Bertino, Guy Lebanon, "Mechanisms for database intrusion detection and response", Data security & privacy, ACM, 2008.

[6] SruthiBandhakavi, "CANDID: Preventing SQL Injection Attacks using Security Dynamic Candidate Evaluations", ACM, 2007.

[7] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications", 33rd ACM SIGPLAN, SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina USA, 2006.

[8] Mehdi Kiani, Andrew Clark,George Mohay, "Evaluation of Anomaly Based Character Distribution Models in the Detection of SQL Injection Attacks", Third International Conference on Availability, Reliability and using Security - Volume 00, , Issue, 4-7 Page(s):47-55, IEEE, March 2008.

[9] David Geer, "Malicious Bots Threaten Network Security", IEEE, Oct 8, 2008.

[10] Y. 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", In Proc. of the 13th Intl. World Wide Web Conference (WWW 04), pages 40-52, May 2004.