A Two Level Intrusion Detection over Multi-Tier Web: Double Guard

Ravipati Ramu [#1], Samsani Surekha [*2]

Department of Computer Science & Engineering,
University College Of Engineering,Vizianagaram ,JNTUK
Vizianagaram Dist,AP,India.

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. One of these attacks is SQL injection, which can give attackers unrestricted access to the databases that underlie Web applications and has become increasingly frequent and serious. This paper presents a new highly automated approach for protecting Web applications against SQL injection that has both conceptual and practical advantages over most existing techniques A Combinational approach for protecting web applications against SQL Injection is discussed in this paper, which is a new idea of incorporating the uniqueness of signature based method and auditing method. From signature based method stand point of view, it presents a detection mode for SQL-Injection using a pair wise sequence alignment of amino acid code formulated from web applications from parameter sent via web server. On the other hand from auditing based method standpoint of view, it analyzes the transaction to find out the malicious access. In Signature based method it uses an approach called Hirschberg algorithm, it is a divide and conquer approach to reduce the time and space complexity. This system was able to stop all of the successful attacks and did not generate any false positives.

Keywords

Security, SQL injection, Hirschberg Algorithm, DBMSAuditing

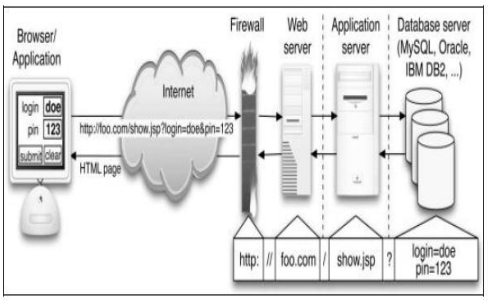
1. Introduction

WEB applications are applications that can be accessed over the Internet by using any compliant Web browser that runs on any operating system and architecture. They have become ubiquitous due to the convenience, flexibility, availability, and interoperability that they provide. Unfortunately, Web applications are also vulnerable to a variety of new security threats. SQL Injection Attacks (SQLIAs) are one of the most significant of such threats [1]. SQLIAs have become increasingly frequent and pose very serious security risks because they can give attackers unrestricted access to the databases that underlie Web applications.

Today's modem web era, expects the organization to concentrate more on web application security. This is the major challenge faced by all the organization to protect their precious data against malicious access or corruptions. Generally the program developers show keen interest in developing the application with usability rather than Incorporating security policy rules. Input validation issue is a security issue if an attacker finds that an application makes unfounded assumptions about the type, length, format, or range of input data. In this paper, we propose a new highly automated approach for dynamic detection and prevention of SQLIAs. Intuitively, our approach works by identifying "trusted" strings in an application and allowing only these trusted strings to be used to create the semantically relevant parts of a SQL query such as keywords or operators. The general mechanism that we use to implement this approach is based on dynamic tainting, which marks and tracks certain data in a program at runtime.

2. Motivation: SQL Injection Attacks

In this section, we first motivate our work by introducing an example of an SQLIA that we use throughout the paper to illustrate our approach and, then, we discuss the main types of SQLIAs in detail. In general, 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 able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings. Interesed readers can refer to the work of Su and Wassermann [2] for a formal definition of SQLIAs. SQLIAs leverage a wide range of mechanisms and input channels to inject malicious commands into a vulnerable application [3]. 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. Fig. 1 shows an example of a typical Web application architecture. In the example, the user interacts with a Web form that takes a login name and pin as inputs and submits them to a Web server. The Web server passes the user supplied credentials to a servlet (show.jsp), which is a special type of Java application that runs on a Web application server and whose execution is triggered by the submission of a URL from a client.



The example servlet, whose code is partially shown in Fig. 2, implements a login functionality that we can find in a typical Web application. It uses input parameters login and pin to dynamically build an SQL query or command. (For simplicity, in the rest of this paper, we use the terms query and command interchangeably.) The login and pin are checked against the credentials stored in the database. If they match, the corresponding user's account information is returned. Otherwise, a null set is returned by the database and the authentication fails. The servlet then uses the response from the database to generate HTML pages that are sent back to the user's browser by the Web server. For this servlet, if a user submits login and pin as "doe "and "123" the application dynamically builds the query:

```
SELECT acct FROM users WHERE login='doe' AND pin=123
```

If login and pin match the corresponding entry in the database, doe's account information is returned and then displayed by function displayAccount(). If there is no match in the database, function sendAuthFailed() displays an appropriate error message. An application that uses this servlet is vulnerable to SQLIAs. For example, if an attacker enters "admin' — " as the username and any value as the pin (for example, "0"), the resulting query is

```
SELECT acct FROM users WHERE login='admin' -- ' AND pin=0
```

In SQL, "--" is the comment operator and everything after it is ignored. Therefore, when performing this query, the database simply searches for an entry where login is equal to admin and returns that database record. After the "successful" login, the function displayAccount () reveals the admin's account information to the attacker. It is important to stress that this example represents an extremely simple kind of attack and we present it for illustrative purposes only. Because simple attacks of this kind are widely used in the literature as examples, they are often mistakenly viewed as the only types of SQLIAs. In reality, there is a wide variety of complex and sophisticated SQL exploits available to attackers. We next discuss the main types of such attacks.

```
1. String login = getParameter("login");
2. String pin = getParameter("pin");
3. Statement stmt = connection.createStatement();
4. String query = "SELECT acct FROM users WHERE login='";
5. query += login + "' AND pin=" + pin;
6. ResultSet result = stmt.executeQuery(query);
7. if (result != null)
8.    displayAccount(result); // Show account
9. else
10.   sendAuthFailed(); // Authentication failed
```

Fig. 2. Excerpt of a Java servlet implementation.

2.1. Tautologies

Tautology-based attacks are among the simplest and best known types of SQLIAs. The general goal of a tautology based attack is to inject SQL tokens that cause the query's conditional statement to always evaluate to true. Although the results of this type of attack are application specific, the most common uses are bypassing authentication pages and extracting data. In this type of injection, an attacker exploits a vulnerable input field that is used in the query's WHERE conditional. This conditional logic is evaluated as the database scans each row in the table. If the conditional represents a tautology, the database matches and returns all of the rows in the table as opposed to matching only one row, as it would normally do in the absence of injection. An example of a tautology-based SQLIA for the servlet in our example in Section 2 is the following:

```
SELECT acct FROM users WHERE login='' OR 1=1 -- ' AND pin=
```

Because the WHERE clause is always true, this query will return account information for all of the users in the database.

2.1.1 Union Queries

Although tautology-based attacks can be successful, for instance, in bypassing authentication pages, they do not give attackers much flexibility in retrieving specific information from a database. Union queries are a more sophisticated type of SQLIA that can be used by an attacker to achieve this goal, in that they cause otherwise legitimate queries to return additional data. In this type of SQLIA, attackers inject a statement of the form "UNION < injected query >." By suitably defining <injected query >, attackers can retrieve information from a specified table. The outcome of this attack is that the database returns a data set that is the union of the results of the original query with the results of the injected query. In our example, an attacker could perform a Union Query injection by injecting the text **" „ 0 UNION SELECT cardNo from CreditCards where acctNo ==7032 -- "** into the login field. The application would then produce the following query:

```
SELECT acct FROM users WHERE login='doe' AND pin=0; drop
table users
```

The original query should return the null set, and the injected query returns data from the "CreditCards" table. In this case, the database returns field "cardNo" for account "7032." The database takes the results of these two queries, unites them, and returns them to the application. In many applications, the effect of this attack would be that the value for "cardNo" is displayed with the account information.

2.1.3 Piggybacked Queries

Similar to union queries, this kind of attack appends additional queries to the original query string. If the attack is successful, the database receives and executes a query string that contains multiple distinct queries. The first query is generally the original legitimate query whereas subsequent queries are the injected malicious queries. This type of attack can be especially harmful because attackers can use it to inject virtually any type of SQL command. In our example, an attacker could inject the text "0; drop table users" into the pin input field

and have the application generate the following query:

```
SELECT acct FROM users WHERE login='doe' AND pin=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 table in the database, which would have the catastrophic consequence of deleting all user information. Other types of queries can be executed using this technique, such as the insertion of new users into the database or the execution of stored procedures. Note that many databases do not require a special character to separate distinct queries, so simply scanning for separators is not an effective way to prevent this attack technique.

3. SQL Injection Attack Types

There are different methods of attacks that depending on the goal of attacker are performed together or sequentially. For a successful SQLIA the attacker should append a syntactically correct command to the original SQL query. Now the following classification of SQLIAs in accordance to the Halfond, Viegas, and Orso researches [4, 5] are presented.

Tautologies:

This type of attack injects SQL tokens to the conditional query statement to be evaluated always true. This type of attack used to bypass authentication control and access to data by exploiting vulnerable input field which use WHERE Clause

"SELECT * FROM employee WHERE userid = '112' and password ='aaa' OR '1'='1'"

As the tautology statement (l = 1) has been added to the query statement so it is always true.

IIIegal/Logically Incorrect Queries:

When a query is rejected, an error message is returned from the database including useful debugging information. This error messages help attacker to find vulnerable parameters in the application and consequently database of the application. In fact attacker injects junk input or SQL tokens in query to produce syntax error, type mismatches, or logical errors by purpose. In this example attacker makes a type mismatch error by injecting the following text into the pin input field:

1) Original URL:

http://www .arch.polimLitieventil?id _ nav=8864

2) SQL Injection:

http://www.arch.polimi.itleventil?id nav=8864'

3) Error message showed:

SELECT name FROM Employee WHERE id =8864\'

From the message error we can fmd out name of table and fields: name; Employee; id. By the gained information attacker can organize more strict attacks.

Union Query:

By this technique, attackers join injected query to the safe query by the word UNION and then can get data about other tables from the application. Suppose for our examples that the query executed from the server is the following:

SELECT Name, Phone FROM Users WHERE Id=$id

By injecting the following Id value:

$id= 1 UNION ALL SELECT creditCardNumber, 1 FROM CreditCarTable

We will have the following query:

SELECT Name, Phone FROM Users WHERE Id= 1
UNION ALL SELECT creditCardNumber, 1 FROM CreditCarTable

Which will join the result of the original query with all the credit card users.

4. Our Approach

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 (typically 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 advantage of the characteristics of SQLIAs and Web applications. First, unlike existing dynamic tainting techniques, our approach is based on the novel 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 evaluation of query strings before they are sent to the database and blocks all queries whose nonliteral parts (that is, SQL keywords and operators) contain one or more characters without trust markings. Finally, our approach has minimal deployment requirements, which makes it both practical and portable. The following sections discuss these key features of our approach in detail.

4.1 Positive Tainting

Positive tainting differs from traditional tainting (hereafter, negative tainting) because it is based on the identification, marking, and tracking of trusted, rather than untrusted, data. This conceptual difference has significant implications for the effectiveness of our approach, in that it helps address problems caused by incompleteness in the identification of relevant data to be marked. Incompleteness, which is one of the major challenges when implementing a security technique based on dynamic tainting, has very different consequences in negative and positive tainting. In the case of negative tainting, incompleteness leads to trusting data that should not be trusted and, ultimately, to false negatives. Incompleteness may thus leave the application vulnerable to attacks and can be very difficult to detect, even after attacks actually occur, because they may go completely unnoticed. With positive tainting, incompleteness may lead to false positives, but it would never result in an SQLIA escaping detection. Moreover, as explained in the following, the false positives generated by our approach, if any, are likely to be detected and easily eliminated early during prerelease testing. Positive tainting uses a white-list, rather than a black-list, policy and follows the general principle of fail-safe defaults, as outlined by Saltzer and Schroeder [6]: In case of incompleteness, positive tainting fails in a way that maintains the security of the system. Fig. 3 shows a graphical depiction of this fundamental difference between negative and positive tainting.

In the context of preventing SQLIAs, the conceptual advantages of positive tainting are especially significant. The way in which Web applications create SQL commands makes the identification of all untrusted data especially problematic and, most importantly, the identification of most trusted data relatively straightforward. Web applications are deployed in many different configurations and interface with a wide range of external systems. Therefore, there are often many potential external untrusted sources of input to be considered for these applications, and enumerating all of them is inherently difficult and error prone. For example, developers initially assumed that only direct user input needed to be marked as tainted. Subsequent exploits demonstrated that additional input sources such as browser cookies and uploaded files also needed to be considered. However, accounting for these additional input sources did not completely solve the problem either. Attackers soon realized the possibility of leveraging local server variables and the database itself as injection sources [7]. In general, it is difficult to guarantee that all potentially harmful data sources have been considered and even

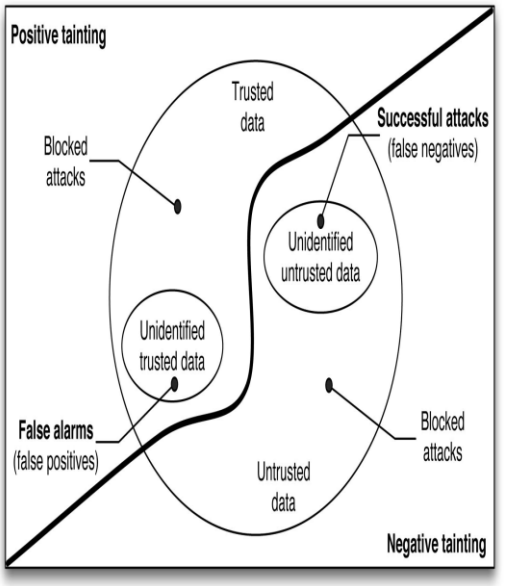


Fig. 3. Identification of trusted and untrusted data.

a single unidentified source could leave the application vulnerable to attacks. The situation is different for positive tainting because identifying trusted data in a Web application is often straightforward and always less error prone. In fact, in most cases, strings hard-coded in the application by developers represent the complete set of trusted data for a Web application.1 This is because it is common practice for developers to build SQL commands by combining hardcoded strings that contain SQL keywords or operators with user-provided numeric or string literals. For Web applications developed this way, our approach accurately and automatically identifies all SQLIAs and generates no false positives. Our basic approach, as explained in the following sections, automatically marks as trusted all hard-coded strings in the code and then ensures that all SQL keywords and operators are built using trusted data. In some cases, this basic approach is not enough because developers can also use external query fragments—partial SQL commands that come from external input sources—to build queries. Because these string fragments are not hardcoded in the application, they would not be part of the initial set of trusted data identified by our approach and the approach would generate false positives when the string fragments are used in a query. To account for these cases, our technique provides developers with a mechanism for specifying sources of external data that should be trusted. The data sources can be of various types such as files, network connections, and server variables.

4.2 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.

4.3 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 untrusted data in SQL commands is not a viable 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 unsanitized 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 untrusted input data in a SQL query as long as the use of such data does not cause an SQLIA. The key feature of syntaxaware evaluation is that it considers the context in which trusted and untrusted 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 untrusted 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. As discussed in Section 4.1, this approach can also handle cases where developers use external query fragments to build SQL commands. In these cases, developers would specify which external data sources must be trusted, and our technique would mark and treat data that comes from these sources accordingly. This default approach, which 1) considers only two kinds of data (trusted and untrusted) and 2) allows only trusted data to form SQL keywords and operators, is adequate for most Web applications. For example, it can handle applications where parts of a query are stored in external files or database records that were created by the developers. Nevertheless, to provide greater flexibility and support a wide range of development practices, our technique also allows developers to associate custom trust markings to different data sources and provide custom trust policies that specify the legal ways in which data with certain trust markings can be used. Trust policies are functions that take as input a sequence of SQL tokens and perform some type of check based on the trust markings associated with the tokens. **BUGZILLA [8] (http://www.bugzilla.org)** is an example of a Web application for which developers might wish to specify a custom trust marking and policy. In BUGZILLA, parts of queries used within the application are retrieved from a database when needed. Of particular concern to developers in this scenario is the potential for second-order injection attacks [9] (that is, attacks that inject into a database malicious strings that result in an SQLIA only when they are later retrieved and used to build SQL queries). In the case of BUGZILLA, the only subqueries that should originate in the database are specific predicates that form a query's WHERE clause. Using our technique, developers could first create a custom trust marking and associate it with the database's data source. Then, they could define a custom trust policy that specifies that data with such a custom trust marking is legal only if it matches a specific pattern, such as ,when applied to subqueries that originate in the database, this policy would allow them to be used only to build conditional clauses that involve the id or severity fields and whose parts are connected using the AND or OR keywords.

5. Conclusion

This paper presented a novel highly automated approach for protecting Web applications from SQLIAs. Our approach consists of 1) identifying trusted data sources and marking data coming from these sources as trusted, 2) using dynamic tainting to track trusted data at runtime, and 3) allowing only trusted data to form the

semantically relevant 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 (rather than untrusted) data in a program. This way, we eliminate the problem of false negatives that may result from the incomplete identification of all untrusted data sources. False positives, although possible in some cases, can typically be easily eliminated during testing. Our approach also provides practical advantages over the many existing techniques whose application requires customized and complex runtime environments: It is defined at the application level, requires no modification of the runtime system, and imposes a low execution overhead.

6. References

[1] "Top Ten Most Critical Web Application Vulnerabilities," OWASP Foundation, http://www.owasp.org/documentation/ topten.html, 2005.

[2] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications.," Proc. Symp.Principles of Programming Languages, pp. 372-382, Jan. 2006.

[3] W.G. Halfond, J. Viegas, and A. Orso, "A Classification of SQLInjection Attacks and Countermeasures," Proc. IEEE Int'l Symp. Secure Software Eng., Mar. 2006.

[4] MUSIC: Mutation-based SQL Injection Vulnerability Checking by Hossain Shahriar and Mohammad Zulkernine, The Eighth International Conference on Quality Software, IEEE- 2008

[5]. SBSQLID: Securing Web Applications with Service Based SQL Injection Detection by Shri.V.Shanmughaneethi, Smt. C.Emilin Shyni, Dr.S.Swamynathan, 2009 International Conference on Advances in Computing, Control, and Telecommunication Technologies.

[6] J. Saltzer and M. Schroeder, "The Protection of Information in Computer Systems," Proc. Fourth ACM Symp. Operating System Principles, Oct. 1973.

[7] C. Anley, "Advanced SQL Injection In SQL Server Applications," white paper, Next Generation Security Software, 2002.

[8] BUGZILLA (http://www.bugzilla.org).

[9] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java," Proc. 21st Ann. Computer Security Applications Conf., pp. 303-311, Dec. 2005.

7. About the Authors

Ravipati Ramu is currently pursuing his M.Tech in Computer Science & Engineering at University College of Engineering, Vizianagaram JNTUK. His area of interests includes Network Security.

Samsani Surekha is currently working as an Assistant Professor in Computer Science and Engineering department, JNTUK University College of Engineering, VZM Dist. Her research interests include Networks, Security, and Data Mining.