PROTECTING WEB APPLICATIONS USING SYNTAX-AWARE EVALUATION

Tejinder Singh*

ABSTRACT

This paper presents a new highly automated move toward for protecting Web applications against SQL injection that has both conceptual and practical advantages over most existing techniques. From a conceptual standpoint, the approach is based on the novel idea of positive tainting and on the concept of syntax-aware evaluation. From a practical standpoint, our technique is precise and efficient, has minimal deployment requirements, and invites a tiny performance overhead in most cases. We have implemented our techniques in the Web Application SQL-injection Preventer (WASP) tool, which we used to perform an practical evaluation on a wide range of Web applications that we subjected to a large and varied set of attacks and genuine accesses. WASP was able to stop all of the otherwise successful attacks and did not generate any false positives.

Keywords: *Security, SQL injection, Runtime, Performance .*

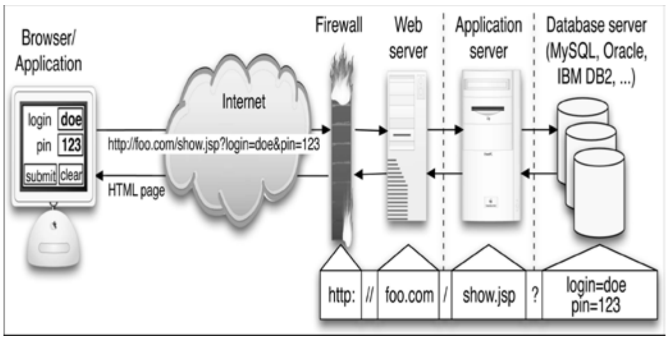
*Research Scholar, Lecturer of computer Science , BFGI, Bathinda, Punjab.

1. INTRODUCTION

Web applications are also vulnerable to a variety of new security threats. SQL Injection Attacks are one of the most significant of such threats .SQL Injections Attacks have become all the time more common and fake as very serious security risks because they can give attackers unrestricted access to the databases that underlie Web applications. For Example Web applications interface with databases that contain information such as customer names, preferences, credit card numbers, purchase orders, and so on. Web applications build SQL queries to access these databases based, in part, on user-provided input. The objective is that Web applications will limit the kinds of queries that can be generated to a safe separation of all possible queries, in any case of what input users provide. However, insufficient input validation can enable attackers to gain complete access to such databases. One way in which this happens is that attackers can submit input strings that contain specially encoded database commands. When the Web application builds a query by using these strings and submits the query to its underlying database, the attacker's embedded commands are executed by the database and the attack succeeds. The results of these attacks are often unsuccessful and can range from leaking of sensitive data (for example, customer data) to the destruction of database contents. In this paper, we also present the results of an extensive empirical evaluation of the effectiveness and efficiency of our technique. To perform this evaluation, we implemented our approach in a tool called Web Application SQL Injection Preventer (WASP) and evaluated WASP on a set of 10 Web applications of various types and sizes. For each application, we protected it with WASP, targeted it with a large set of attacks and legal accesses, and assessed the ability of our technique to detect and prevent attacks without stopping legal accesses. The results of the evaluation are promising. Our technique was able to stop all of the attacks without generating false positives for any of the legitimate accesses. Moreover, our technique proved to be efficient, imposing only a low overhead on the Web applications.

2. INSPIRATION: SQL INJECTION ATTACKS

In general, SQL Injection Attackers are a class of code injection attacks that take advantage of the lack of validation of user input. These attacks occur when developers hard-coded strings with user-provided input to create dynamic queries. Naturally, 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 expertise input strings. SQL Injection Attacker influence 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 control 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. Fig 1



2.1 Main Variants of SQL Injection

Attacks

Over the past several years, attackers have developed a wide array of sophisticated attack techniques that can be used to exploit SQL injection vulnerabilities. For example, developers and researchers often assume that SQLIAs are introduced only via user input that is submitted as part of a Web form. This assumption misses the fact that any external input that is used to build a query string may represent a possible channel for SQLIAs. In fact, it is common to see other external sources of input such as fields from an HTTP cookie or server variables used to build a query. Since cookie values are under the control of the user's browser and server variables are often set using values from HTTP headers, these values are actually external strings that can be manipulated by an attacker. Depending on the type and extent of the vulnerability, the results of these attacks can include crashing the database, gathering information about the tables in the database schema, establishing covert channels, and open-ended injection of virtually any SQL command. Here, we implementation. summarize the main techniques for performing SQL Injection Attackers. We provide additional information and examples of how these techniques work in.

Tautologies : 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.

Union Queries : 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 SQL Injection Attacker 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 SQL Injection attackers inject a statement of the form "UNION < injected query > ." By suitably defining < injected query > , attackers can retrieve information from a specified table.

Piggybacked Queries : Similar to union queries, this kind of attack appends additional queries to the original query string. 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".

Alternate Encoding : Many types of SQLIAs involve the use of special characters such as single quotes, dashes, or semicolons as part of the inputs to a Web application. Therefore, basic protection techniques against these attacks check the input for the presence of such characters and escape them or simply block inputs that contain them. Alternate encodings let attackers modify their injected strings in a way that avoids these typical signature-based and filter-based checks. Encodings such as ASCII, hexadecimal, and Unicode can be used in conjunction with other techniques to allow an attack to escape straightforward detection approaches that simply scan for certain known "bad characters."

3. SYNTAX-AWARE EVALUATION

Simply forbidding the use of entrusted 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 infected input as long as it has been processed by a washing 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 passable. 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 entrusted 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 entrusted 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 entrusted 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 3.1, 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.

4. OUR IMPLEMENTATION: WASP

To evaluate our approach, we developed a prototype tool called WASP (Web Application SQL-injection Preventer), which is written in

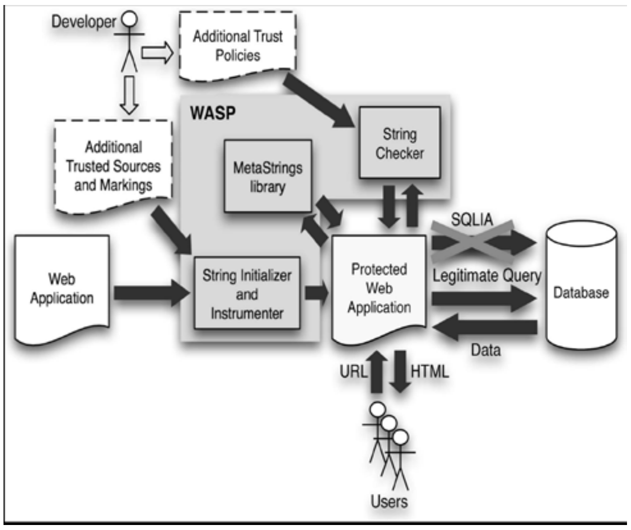

Developer
Additional Trust Policies
WASP
String Checker
Additional Trusted Sources and Markings
MetaStrings library
SQLIA
Web Application
String Initializer and Instrumenter
Protected Web Application
Legitimate Query
Database
Data
URL
HTML
Users

Fig 2. High-level overview of the approach and tool

Java and implements our technique for Java-based Web applications. We target Java because it is one of the most commonly used languages for Web applications. Fig. 2 shows the high-level architecture of WASP. As this figure shows, WASP consists of a library (MetaStrings) andtwo core modules (STRING INITIALIZER AND INSTRUMENTER and STRING CHECKER). The MetaStrings library provides functionality for assigning trust markings to strings and precisely propagating the markings at runtime. Module STRING INITIALIZER AND INSTRUMENTER instruments Web applications to enable the use of the MetaStrings library and adds calls to the STRING CHECKER module. Module STRING CHECKER performs syntax-aware evaluation of query strings right before the strings are sent to the database. In the next sections, we discuss WASP's modules in more detail. We use the sample code introduced in Section 2 to provide examples of various implementation aspects.

4.1 The MetaStrings Library

MetaStrings is our library of classes that mimic and extend the behavior of Java's standard string classes (that is, Character, String, StringBuilder, and String Buffer).2 For each string class C, MetaStrings provides a "meta" version of the class MetaC, which has the same functionality as C, but allows for associating metadata with
each character in a string and tracking the metadata as the string is manipulated at runtime. The Meta Strings library takes advantage of the object oriented features of the Java language to provide complete mediation of string operations that could affect string values and their associated trust markings. Encapsulation and information hiding guarantee that the internal representation of a string class is accessed only through the class's interface. Polymorphism and dynamic binding let us add functionality to a string class by creating a subclass that overrides relevant methods of the original class and replacing instantiations of the original class with instantiations of the subclass. In our implementation, we leverage the object-oriented features of Java.

5. CONCLUSION

This paper 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, This way, we eliminate the problem of false negatives that may result from the incomplete identification of all entrusted 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. 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 accesses. WASP successfully and efficiently stopped over 12,000 attacks without generating any false positives. Both our tool and the experimental infrastructure are available to other researchers. We have two immediate goals for future work. First, we
will extend our experimental results by using WASP to protect actually deployed Web applications. Our first target will be a set of Web applications that run at Georgia Tech. This will allow us to assess the effectiveness of WASP in real settings and also to collect a valuable set of real legal accesses and, possibly, attacks. Second, we will implement the approach for binary applications. We have already started developing the infrastructure to perform tainting at the binary level and developed a proof-of-concept prototype .

REFERENCE

[1] C. Anley, “Advanced SQL Injection In SQL Server Applications,” white paper, Next Generation Security Software, 2002.

[2] S.W. Boyd and A.D. Keromytis, “SQLrand: Preventing SQL Injection Attacks,” Proc. Second Int’l Conf. Applied Cryptography and Network Security, pp. 292-302, June 2004.

[3] W.R. Cook and S. Rai, “Safe Query Objects: Statically Typed Objects as Remotely Executable Queries,” Proc. 27th Int’l Conf. Software Eng., pp. 97-106, May 2005.

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

[5] 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,” Proc. 13th Int’l Conf. World Wide Web, pp. 40-52, May 2004.

[6] Z. Su and G. Wassermann, “The Essence of Command Injection Attacks in Web Applications.,” Proc. 33rd Ann. Symp. Principles of Programming Languages, pp. 372-382, Jan. 2006.

[7] Y. Xie and A. Aiken, “Static Detection of Security Vulnerabilities
in Scripting Languages,” Proc. 15th Usenix Security Symp., Aug. 2006.

[8] R. McClure and I. Kru¨ ger, “SQL DOM: Compile Time Checking of Dynamic SQL Statements,” Proc. 27th Int’l Conf. Software Eng., pp. 88-96, May 2005.