Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks

William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios
College of Computing – Georgia Institute of Technology
{whalfond, orso, manolios}@cc.gatech.edu

ABSTRACT

SQL injection attacks pose a serious threat to the security of Web applications because they can give attackers unrestricted access to databases that contain sensitive information. In this paper, we propose a new, highly automated approach for protecting existing Web applications against SQL injection. Our approach has both conceptual and practical advantages over most existing techniques. From the conceptual standpoint, the approach is based on the novel idea of positive tainting and the concept of syntax-aware evaluation. From the practical standpoint, our technique is at the same time precise and efficient and has minimal deployment requirements. The paper also describes WASP, a tool that implements our technique, and a set of studies performed to evaluate our approach. In the studies, we used our tool to protect several Web applications and then subjected them to a large and varied set of attacks and legitimate accesses. The evaluation was a complete success: WASP successfully and efficiently stopped all of the attacks without generating any false positives.

Categories and Subject Descriptors: D.2.0 [Software Engineering]: General—*Protection mechanisms*;

General Terms: Security

Keywords: SQL injection, dynamic tainting, runtime monitoring

1. INTRODUCTION

SQL injection attacks (SQLIAs) are one of the major security threats for Web applications [5]. Successful SQLIAs can give attackers access to and even control of the databases that underly Web applications, which may contain sensitive or confidential information. Despite the potential severity of SQLIAs, many Web applications remain vulnerable to such attacks.

In general, SQL injection vulnerabilities are caused by inadequate input validation within an application. Attackers take advantage of these vulnerabilities by submitting input strings that contain specially-encoded database commands to the application. When the application builds a query 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. Although this general mechanism is well understood, straightforward solutions based on defensive coding practices have been less than successful for several reasons. First, it is difficult to implement and enforce a rigorous defensive coding discipline. Second, many solutions based on defensive coding address only a subset of the possible attacks. Finally, defensive coding is problematic in the case of legacy software because of the cost and complexity of retrofitting existing code. Researchers have proposed a wide range of alternative techniques to address SQLIAs, but many of these solutions have limitations that affect their effectiveness and practicality.

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 certain parts of an 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.

The kind of dynamic tainting we use gives our approach several important advantages over techniques based on different mechanisms. Many techniques rely on complex static analyses in order to find potential vulnerabilities in code (*e.g.*, [9, 15, 26]). These kinds of conservative static analyses can generate high rates of false positives or may have scalability issues when applied to large, complex applications. Our approach does not rely on complex static analyses and is very efficient and precise. Other techniques involve extensive human effort (*e.g.*, [4, 18, 24]). They require developers to manually rewrite parts of their applications, build queries using special libraries, or mark all points in the code at which malicious input could be introduced. In contrast, our approach is highly automated and in most cases requires minimal or no developer intervention. Lastly, several proposed techniques require the deployment of extensive infrastructure or involve complex configurations (*e.g.*, [2, 23, 25]). Our approach does not require additional infrastructure and can be deployed automatically.

Compared to other existing techniques based on dynamic tainting (*e.g.*, [8, 20, 21]), our approach makes several conceptual and practical improvements that take advantage of the specific characteristics of SQLIAs. The *first conceptual advantage* of our approach is the use of positive tainting. Positive tainting identifies and tracks trusted data, whereas traditional ("negative") tainting focuses on untrusted data. In the context of SQLIAs, there are several reasons why positive tainting is more effective than negative tainting. First, in Web applications, trusted data sources can be more easily and accurately identified than untrusted data sources; therefore, the use of positive tainting leads to increased automation.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.

Second, the two approaches differ significantly in how they are affected by incompleteness. With negative tainting, failure to identify the complete set of untrusted data sources would result in false negatives, that is, successful undetected attacks. With positive tainting, conversely, missing trusted data sources would result in false positives, which are undesirable, but whose presence can be detected immediately and easily corrected. In fact, we expect that most false positives would be detected during pre-release testing. The *second conceptual advantage* of our approach is the use of flexible syntax-aware evaluation, which gives developers a mechanism to regulate the usage of string data based not only on its source, but also on its syntactical role in a query string. In this way, developers can use a wide range of external input sources to build queries, while protecting the application from possible attacks introduced via these sources.

The practical advantages of our approach are that it imposes a low overhead on the application and has minimal deployment requirements. Efficiency is achieved by using a specialized library, called MetaStrings, that accurately and efficiently assigns and tracks trust markings at runtime. The only deployment requirements for our approach are that the Web application must be instrumented and deployed with our MetaStrings library, which is done automatically. The approach does not require any customized runtime system or additional infrastructure.

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 WASP (Web Application SQL-injection Preventer) and evaluated WASP on a set of seven Web applications of various types and sizes. For each application, we protected it with WASP, targeted it with a large set of attacks and legitimate accesses, and assessed the ability of our technique to detect and prevent attacks without stopping legitimate 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.
The main contributions of this work are:

- A new, automated technique for preventing SQLIAs based on the novel concept of positive tainting and on flexible syntax-aware evaluation.
- A mechanism to perform efficient dynamic tainting of Java strings that precisely propagates trust markings while strings are manipulated at runtime.
- A tool that implements our SQLIA prevention technique for Java-based Web applications and has minimal deployment requirements.
- An empirical evaluation of the technique that shows its effectiveness and efficiency.

The rest of this paper is organized as follows. In Section 2, we introduce SQLIAs with an example that is used throughout the paper. Sections 3 and 4 discuss the approach and its implementation. Section 5 presents the results of our evaluation. We discuss related work in Section 6 and conclude in Section 7.

2. SQL INJECTION ATTACKS

Intuitively, an SQL Injection Attack (SQLIA) occurs when an attacker changes the developer's intended structure of an SQL command by inserting new SQL keywords or operators. (Su and Wassermann provide a formal definition of SQLIAs in [24].) SQLIAs leverage a wide range of mechanisms and input channels to inject

```
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
```

Figure 1: Excerpt of a Java servlet implementation.

malicious commands into a vulnerable application [10]. In this section we introduce an example application that contains an SQL injection vulnerability and show how an attacker can leverage the vulnerability to perform an SQLIA. Note that the example represents an extremely simple kind of attack, and we present it for illustrative purposes only. Interested readers may refer to References [1] and [10] for further examples of the different types of SQLIAs.

The code excerpt in Figure 1 represents the implementation of login functionality that we can find in a typical Web application. This type of login function would commonly be part of a Java *servlet*, a type of Java application that runs on a Web application server, and whose execution is triggered by the submission of a URL from a user of the Web application. The servlet in the example uses the input parameters `login` and `pin` to dynamically build an SQL query or command.[1] 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 the Web server.

Given the servlet code, 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 user name and any value as the pin (*e.g.*, "`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()` would therefore reveal the `admin`'s account information to the attacker.

3. OUR APPROACH

Our approach is based on dynamic tainting, which has been widely 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. First, unlike any existing dynamic tainting techniques that we are aware of, our ap-

[1]For simplicity, in the rest of this paper we use the terms query and command interchangeably.

proach is based on the novel concept of *positive tainting*—the identification and marking of trusted instead of untrusted data. Second, our approach performs *accurate 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 non-literal parts (*i.e.*, 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 the key features of our approach in detail.

3.1 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 occur. With positive tainting, incompleteness may lead to false positives, but never results in an SQLIA escaping detection. Moreover, as explained below, the false positives generated by our approach are likely to be detected and easily eliminated early, during pre-release testing. Positive tainting follows the general principle of *fail-safe defaults* as outlined by Saltzer and Schroeder in [22]: in case of incompleteness, positive tainting fails in a way that maintains the security of the system.

In the context of preventing SQLIAs, these 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 all 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 [1]. In general, it is difficult to guarantee that all potentially harmful data sources have been considered, and even 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.[2] The reason for this is that it is common practice for developers to build SQL commands by combining hard-coded strings that contain SQL keywords or operators with user-provided numeric or string literals. For Web applications developed in this way, which includes the applications used in our empirical evaluation, 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 coming from external input sources—to build queries. Because these string fragments are not hard-coded 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 to specify additional sources of external data that should be trusted. The data sources can be of various types, such as files, network connections, and server variables. Our approach uses this information to mark data coming from these additional sources as trusted.

In a typical scenario, we expect developers to specify most of the trusted sources beforehand. However, some of these sources might be overlooked until after a false positive is reported, in which case developers would add the omitted data source to the list of trusted sources. In this process, the set of trusted data sources grows monotonically and eventually converges to a complete set that produces no false positives. It is important to note that false positives that occur after deployment would be due to the use of external data sources that have never been used during in-house testing. In other words, false positives are likely to occur only for totally untested parts of the application. Therefore, even when developers fail to completely identify and mark additional sources of trusted input beforehand, we expect these sources to be identified during normal testing of the application, and the set of trusted data to quickly converge to the complete set.

3.2 Accurate Taint Propagation

Taint propagation consists of tracking taint markings associated with the data while the data is used and manipulated at runtime. When tainting is used for security-related applications, it is especially important for the propagation to be accurate. Inaccurate propagation can undermine the effectiveness of a technique by associating incorrect markings to data, which would cause the data to be mishandled. In our approach, we provide a mechanism to accurately mark and propagate taint information by (1) tracking taint markings at a low level of granularity and (2) precisely accounting for the effect of functions that operate on the tainted data.

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.

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 (*i.e.*, 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, and 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.

[2] We assume that developers are trustworthy. An attack encoded by a developer would not be an SQLIA but a form of back-door attack, which is not the problem addressed in this paper.

We discuss the language specific details of our implementation of the taint markings and their propagation in Section 4.

3.3 Syntax-Aware Evaluation

Besides 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. An approach that simply forbids 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 sub-string 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 in fact 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 depend 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 an 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 untrusted data is used to make sure that all parts of a query other than string or numeric literals (*e.g.*, 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 (*e.g.*, if an SQL operator contains characters not marked as trusted), we can assume that the operator has been injected by an attacker and block the query.

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 an 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 (*i.e.*, substrings) other than literals contain only trusted data. If all of the tokens pass this check, the query is considered safe and allowed to execute. 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 coming 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 (`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 [1] (*i.e.*, 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 sub-queries that should originate from 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 custom trust marking are legal only if they match a specific pattern, such as the following:

```
(id|severity)='\w+' ((AND|OR) (id|severity)='\w+')*
```

When applied to sub-queries originating from 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.

3.4 Minimal Deployment Requirements

Most existing approaches based on dynamic tainting require the use of customized runtime systems and/or impose a considerable overhead on the protected applications (see Section 6). On the contrary, our approach has minimal deployment requirements and is efficient, which makes it practical for usage in real settings. The use of our technique does not necessitate a customized runtime system. It requires only minor, localized instrumentation of the application to (1) enable the usage of our modified string library and (2) insert the calls that perform syntax-aware evaluation of a query before the query is sent to the database. The protected application is then deployed as any normal Web application, except that the deployment must include our string library. Both instrumentation and deployment are fully automated. We discuss deployment requirements and overhead of the approach in greater detail in Sections 4.5 and 5.3.

4. IMPLEMENTATION

To evaluate our approach, we developed a prototype tool called WASP (Web Application SQL Injection Preventer) that is written in Java and implements our technique for Java-based Web applications. We chose to target Java because it is a commonly-used language for developing Web applications. Moreover, we already have a significant amount of analysis and experimental infrastructure for Java applications. We expect our approach to be applicable to other languages as well.

Figure 2 shows the high-level architecture of WASP. As the figure shows, WASP consists of a library (MetaStrings) and two 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 add 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 illustrative examples of various implementation aspects.

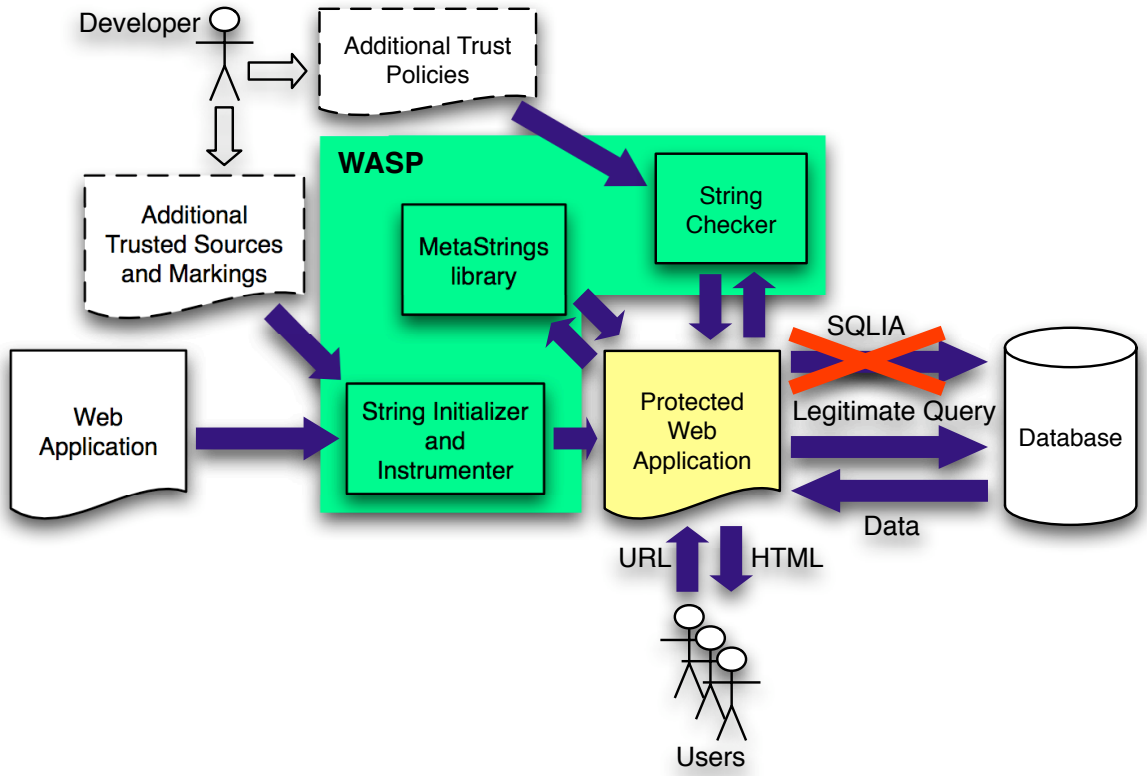

Figure 2: High-level overview of the approach and tool.

4.1 The MetaStrings Library

MetaStrings is our library of classes that mimic and extend the behavior of Java's standard string classes (*i.e.*, `Character`, `String`, `StringBuilder`, and `StringBuffer`).[3] For each string class `C`, MetaStrings provides a "meta" version of the class, `MetaC`, that 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 MetaStrings 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 (1) creating a subclass that overrides all methods of the original class and (2) replacing instantiations of the original class with instantiations of the subclass.

As an example, Figure 3 shows an intuitive view of the MetaStrings class that corresponds to Java's `String` class. As the figure shows, `MetaString` extends class `String`, has the same internal representation, and provides the same methods. `MetaString` also contains additional data structures for storing metadata and associating the metadata with characters in the string. Each method of class `MetaString` overrides the corresponding method in `String`, providing the same functionality as the original method, but also updating the metadata based on the method's semantics. For example, a call to method `substring(2,4)` on an object `str` of class `MetaString` would return a new `MetaString` that contains the second and third characters of `str` and the corresponding metadata. In addition to the overridden methods, MetaStrings classes also provide methods for setting and querying the metadata associated with a string's characters.

The use of MetaStrings has the following benefits: (1) it allows for associating trust markings at the granularity level of single characters; (2) it accurately maintains and propagates trust markings; (3) it is defined completely at the application level and therefore does not require a customized runtime system; (4) its usage requires only minimal and automatically performed changes to the application's bytecode; and (5) it imposes a low execution overhead on the Web application (See Section 5.3).

The main limitations of the current implementation of the MetaStrings library are related to the handling of primitive types, native methods, and reflection. MetaStrings cannot currently assign trust markings to primitive types, so it cannot mark `char` values. Because we do not instrument native methods, if a string class is passed as an argument to a native method, the trust marking associated with the string might not be correct after the call. In the case of hard-coded strings created through reflection (by invoking a string constructor by name), our instrumenter for MetaStrings would not recognize the constructors and would not change these instantiations to instantiations of the corresponding meta classes. However, the MetaStrings library can handle most other uses of reflection, such as invocation of string methods by name.

In practice, these limitations are of limited relevance because they represent programming practices that are not normally used to build SQL commands (e.g., representing strings using primitive `char` values). Moreover, during instrumentation of a Web application, we identify and report these potentially problematic situations to the developers.

4.2 Initialization of Trusted Strings

To implement positive tainting, WASP must be able to identify and mark trusted strings. There are three categories of strings that

[3]For simplicity, hereafter we use the term string to refer to all string-related classes and objects in Java.

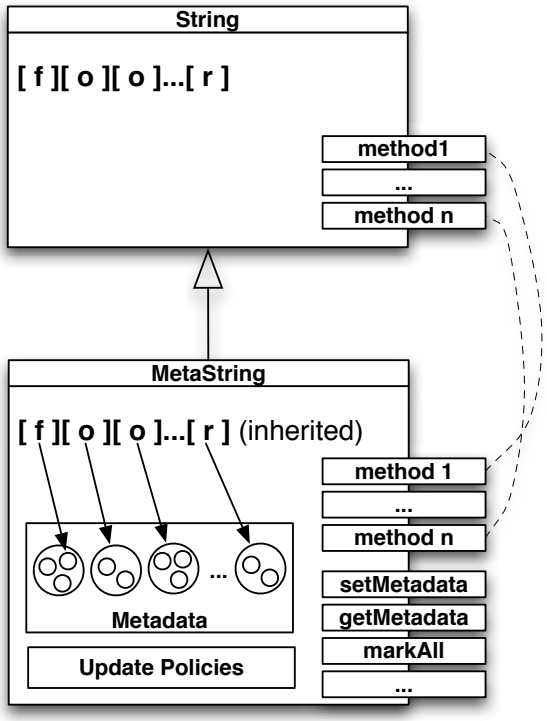

Figure 3: Intuitive view of a MetaStrings library class.

WASP must consider: hard-coded strings, strings implicitly created by Java, and strings originating from external sources. In the following sections, we explain how strings from each category are identified and marked.

Hard-Coded Strings. The identification of hard-coded strings in an application's bytecode is a fairly straightforward process. In Java, hard-coded strings are represented using `String` objects that are created automatically by the Java Virtual Machine (JVM) when string literals are loaded onto the stack. (The JVM is a stack-based interpreter.) Therefore, to identify hard-coded strings, WASP simply scans the bytecode and identifies all load instructions whose operand is a string constant. WASP then instruments the code by adding, after each of these load instructions, code that creates an instance of a `MetaString` class using the hard-coded string as an initialization parameter. Finally, because hard-coded strings are completely trusted, WASP adds to the code a call to the method of the newly created `MetaString` object that marks all characters as trusted. At runtime, polymorphism and dynamic binding allow this instance of the MetaString object to be used in any place where the original `String` object would have been used.

Figure 4 shows an example of this bytecode transformation. The Java code at the top of the figure corresponds to line 4 of our servlet example (see Figure 1), which creates one of the hard-coded strings in the servlet. Underneath, we show the original bytecode (left), and the modified bytecode (right). The modified bytecode contains additional instructions that (1) load a new `MetaString` object on the stack, (2) call the `MetaString` constructor using the previous string as a parameter, and (3) call the method `markAll`, which assigns the given trust marking to all characters in the string.

Implicitly-Created Strings. In Java programs, the creation of some string objects is implicitly added to the bytecode by the compiler. For example, Java compilers typically translate the string concatenation operator ("+") into a sequence of calls to the `append` method of a newly-created `StringBuilder` object. WASP must replace these string objects with their corresponding MetaStrings objects so that they can maintain and propagate the trust markings of the strings on which they operate. To do this, WASP scans the bytecode for instructions that create new instances of the string classes used to perform string manipulation and modifies each such instruction so that it creates an instance of the corresponding MetaStrings class instead. In this case, WASP does not associate any trust markings with the newly-created MetaStrings objects. These objects are not trusted per se, and they become marked only if the actual values assigned to them during execution are marked.

Figure 5 shows the instrumentation added by WASP for implicitly-created strings. The Java source code corresponds to line 5 in our example servlet. The `StringBuilder` object at offset 28 in the original bytecode is added by the Java compiler when translating the string concatenation operator ("+"). WASP replaces the instantiation at offset 28 with the instantiation of a `MetaStringBuilder` class and then changes the subsequent invocation of the constructor at offset 37 so that it matches the newly instantiated class. Because `MetaStringBuilder` extends `StringBuilder`, the subsequent calls to the append method invoke the correct method in the `MetaStringBuilder` class.

Strings from External Sources. To use query fragments coming from external (trusted) sources, developers must list these sources in a configuration file that WASP processes before instrumenting the application. The specified sources can be of different types, such as files (specified by name), network connections (specified by host and port), and databases (specified by database name, table, field, or combination thereof). For each source, developers can either specify a custom trust marking or use the default trust marking (the same used for hard-coded strings). WASP uses the information in the configuration file to instrument the external trusted sources according to their type.

To illustrate this process, we describe the instrumentation that WASP performs for trusted strings coming from a file. In the configuration file, the developer specifies the name of the file (*e.g.*, `foo.txt`) as a trusted source of strings. Based on this information, WASP scans the bytecode for all instantiations of new file objects (*i.e.*, `File`, `FileInputStream`, `FileReader`) and adds instrumentation that checks the name of the file being accessed. At runtime, if the name of the file matches the name(s) specified by the developer (`foo.txt` in this case), the file object is added to an internal list of currently trusted file objects. WASP also instruments all calls to methods of file-stream objects that return strings, such as `BufferedReader`'s `readLine` method. At runtime, the added code checks to see whether the object on which the method is called is in the list of currently trusted file objects. If so, it marks the generated strings with the trust marking specified by the developer for the corresponding source.

We use a similar strategy to mark network connections. In this case, instead of matching file names at runtime, we match hostnames and ports. The interaction with databases is more complicated and requires WASP not only to match the initiating connection, but also to trace tables and fields through instantiations of the `Statement` and `ResultSet` objects created when querying the database.

Instrumentation Optimization. Our current instrumentation approach is conservative and may generate unneeded instrumentation. We could limit the amount of instrumentation inserted in the code by leveraging static information about the program. For example, data-flow analysis could identify strings that are not involved

Source Code: `4. String query = "SELECT acct FROM users WHERE login='";`

Original Bytecode	Modified Bytecode

`24. ldc "SELECT acct FROM users WHERE login='"`	`24a. new MetaString`
	`24b. dup`
	`24c. ldc "SELECT acct FROM users WHERE login='"`
	`24e. invokespecial MetaString.<init>:(LString)V`
	`24d. iconst_1`
	`24e. invokevirtual MetaString.markAll:(I)V`

Figure 4: Instrumentation for hard-coded strings.

Source Code: `5. query += login + "' AND pin=" + pin;`

Original Bytecode	Modified Bytecode

`28. new StringBuilder`
`31. dup`
`32. aload 4`
`34. invokestatic String.valueOf:(Object)LString;`
`37. invokespecial StringBuilder.<init>:(LString;)V`
`40. aload_1`
`41. invokevirtual StringBuilder.`
`append:(LString;)LStringBuilder;`
`44. ldc "' AND pin="`
`46. invokevirtual StringBuilder.`
`append:(LString;)LStringBuilder;`
`49. aload_2`
`50. invokevirtual StringBuilder.`
`append:(LString;)LStringBuilder;`
`53. invokevirtual StringBuilder.toString:()LString;`
	`28. new MetaStringBuilder`
	`31. dup`
	`32. aload 4`
	`34. invokestatic String.valueOf:(LObject)LString;`
	`37. invokespecial MetaStringBuilder.<init>:(LString;)V`
	`40. aload_1`
	`41. invokevirtual StringBuilder.append:(LString;)LStringBuilder;`
	`44a. new MetaString`
	`44b. dup`
	`44c. ldc "' AND pin="`
	`44e. invokespecial MetaString.<init>:(LString)V`
	`44d. iconst_1`
	`44e. invokevirtual MetaString.markAll:(I)V`
	`46. invokevirtual StringBuilder.append:(LString;)LStringBuilder;`
	`49. aload_2`
	`50. invokevirtual StringBuilder.append:(LString;)LStringBuilder;`
	`53. invokevirtual StringBuilder.toString:()LString;`

Figure 5: Instrumentation for implicitly-created strings.

with the construction of query strings and thus do not need to be instrumented. Another example involves cases where static analysis could determine that the filename associated with a file object is never one of the developer-specified trusted filenames, that object would not need to be instrumented. Analogous optimizations could be implemented for other external sources. We did not incorporate any of these optimizations in the current tool because we were mostly interested in having an initial prototype to assess our technique. However, we are planning to implement them in future work to further reduce runtime overhead.

4.3 Handling False Positives

As discussed in Section 3, sources of trusted data that are not specified by the developers beforehand would cause WASP to generate false positives. To assist the developers in identifying data sources that they initially overlooked, WASP provides a special mode of operation, called "learning mode", that would typically be used during in-house testing. When in learning mode, WASP adds an additional unique taint marking to *each* string in the application. Each marking consists of an ID that maps to the fully qualified class name, method signature, and bytecode offset of the instruction that instantiated the corresponding string.

If WASP detects an SQLIA while in learning mode, it uses the markings associated with the untrusted SQL keywords and operators in the query to report the instantiation point of the corresponding string(s). If the SQLIA is actually a false positive, knowing the position in the code of the offending string(s) would help developers correct omissions in the set of trusted inputs.

4.4 Syntax-Aware Evaluation

The STRING CHECKER module performs syntax-aware evaluation of query strings and is invoked right before the strings are sent to the database. To add calls to the STRING CHECKER module, WASP first identifies all of the *database interaction points*: points in the application where query strings are issued to an underlying database. In Java, all calls to the database are performed via specific methods and classes in the JDBC library (`http://java.sun.com/products/jdbc/`). Therefore, these points can be identified through a simple matching of method signatures. After identifying the database interaction points, WASP inserts a call to the syntax-aware evaluation function, `MetaChecker`, immediately before each interaction point. `MetaChecker` takes the MetaStrings object that contains the query about to be executed as a parameter.

When invoked, `MetaChecker` processes the SQL string about to be sent to the database as discussed in Section 3.3. First, it tokenizes the string using an SQL parser. Ideally, WASP would use a database parser that recognizes the exact same dialect of SQL that is used by the database. This would guarantee that WASP interprets the query in the same way as the database and would prevent attacks based on alternate encodings [1]—attacks that obfuscate keywords and operators to elude signature-based checks. Our current implementation includes parsers for SQL-92 (ANSI) and PostgreSQL. After tokenizing the query string, `MetaChecker` enforces the default trust policy by iterating through the tokens that correspond to keywords and operators and examining their trust markings. If any of these tokens contains characters that are not marked as trusted, the query is blocked and reported.

If developers specified additional trust policies, `MetaChecker` invokes the corresponding checking function(s) to ensure that the query complies with them. In our current implementation, trust policies are developer-defined functions that take the list of SQL tokens as input, perform some type of check on them based on their trust markings, and return a `true` or `false` value depending on the outcome of the check. Trust policies can implement functionality that ranges from simple pattern matching to sophisticated checks that use externally-supplied contextual information. If all custom trust policies return a positive outcome, WASP allows the query to be executed on the database. Otherwise, it classifies the query as an SQLIA, blocks it, and reports it.

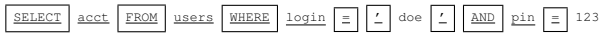

Figure 6: Example query 1 after parsing by runtime monitor.

Figure 7: Example query 2 after parsing by runtime monitor.

We illustrate how the default policy for syntax-aware evaluation works using our example servlet and the legitimate and malicious query examples from Section 2. For the servlet there are no external sources of strings or additional trust policies, so WASP only marks the hard-coded strings as trusted, and only the default trust policy is applied. Figure 6 shows the sequence of tokens in the legitimate query as they would be parsed by `MetaChecker`. In the figure, SQL keywords and operators are surrounded by boxes. The figure also shows the trust markings associated with the strings, where an underlined character is a character with full trust markings. Because the default trust policy is that all keyword and operator tokens must have originated from trusted strings, `MetaChecker` simply checks whether all these tokens are comprised of trusted characters. The query in Figure 6 conforms to the trust policy and is thus allowed to execute on the database.

Consider the malicious query, where the attacker submits "`admin'--`" as the login and "`0`" as the pin. Figure 7 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 `MetaChecker` would find that the last two tokens, `'` and `--` contain untrusted characters. It would therefore classify the query as an SQLIA and prevent it from executing.

4.5 Deployment Requirements

Using WASP to protect a Web application requires the developer to run an instrumented version of the application. There are two general implementation strategies that we can follow for the instrumentation: off-line or on-line. Off-line instrumentation instruments the application statically and deploys the instrumented version of the application. On-line instrumentation deploys an unmodified application and instruments the code at load time (*i.e.*, when classes are loaded by the JVM). This latter option allows for a great deal of flexibility and can be implemented by leveraging the new instrumentation package introduced in Java 5 (`http://java.sun.com/j2se/1.5.0/`).

Unfortunately, the current implementation of the Java 5 instrumentation package is still incomplete and does not yet provide some key features needed by WASP. In particular, it does not allow for clearing the `final` flag in the string library classes, which prevents the MetaStrings library from extending them. Because of this limitation, for now we have chosen to rely on off-line instrumentation and to splice into the Java library a version of the string classes in which the `final` flag has been cleared.

Overall, the deployment requirements for our approach are fairly lightweight. The modification of the Java library is performed only once, in a fully automated way, and takes just a few seconds. No modification of the Java Virtual Machine is required. The instrumentation of a Web application is also performed automatically. Given the original application, WASP creates a deployment archive that contains the instrumented application, the MetaStrings library, and the string checker module. At this point, the archive can be deployed like any other Web application. WASP can therefore be easily and transparently incorporated into an existing build process.

Table 1: Subject programs for the empirical study.

Subject	*LOC*	*DBIs*	*Servlets*	*Params*
Checkers	5,421	5	18 (61)	44 (44)
Office Talk	4,543	40	7 (64)	13 (14)
Employee Directory	5,658	23	7 (10)	25 (34)
Bookstore	16,959	71	8 (28)	36 (42)
Events	7,242	31	7 (13)	36 (46)
Classifieds	10,949	34	6 (14)	18 (26)
Portal	16,453	67	3 (28)	39 (46)

5. EVALUATION

The goal of our empirical evaluation is to assess the effectiveness and efficiency of the approach presented in this paper when applied to a testbed of Web applications. In the evaluation, we used our implementation of WASP and investigated the following three research questions:

RQ1: What percentage of attacks can WASP detect and prevent that would otherwise go undetected and reach the database?

RQ2: What percentage of legitimate accesses does WASP identify as SQLIAs and prevent from executing on the database?

RQ3: How much runtime overhead does WASP impose?

The first two questions deal with the *effectiveness* of the technique: RQ1 addresses the false negative rate of the technique, and RQ2 addresses the false positive rate. RQ3 deals with the *efficiency* of the proposed technique. The following sections discuss our experiment setup, protocol, and results.

5.1 Experiment Setup

Our experiments are based on an evaluation framework that we developed and has been used by us and other researchers in previous work [9, 24]. The framework provides a testbed that consists of several Web applications, a logging infrastructure, and a large set of test inputs containing both legitimate accesses and SQLIAs. In the next two sections we summarize the relevant details of the framework.

5.1.1 Subjects

Our set of subjects consists of seven Web applications that accept user input via Web forms and use it to build queries to an underlying database. Five of the seven applications are commercial applications that we obtained from GotoCode (`http://www.gotocode.com/`): Employee Directory, Bookstore, Events, Classifieds, and Portal. The other two, Checkers and OfficeTalk, are applications developed by students that have been used in previous related studies [7].

For each subject, Table 1 provides the size in terms of lines of code (*LOC*) and the number of database interaction points (*DBIs*). To be able to perform our studies in an automated fashion and collect a larger number of data points, we considered only those servlets that can be accessed directly, without complex interactions with the application. Therefore, we did not include in the evaluation servlets that require the presence of specific session data (*i.e.*, cookies containing specific information) to be accessed. Column *Servlets* reports, for each application, the number of servlets considered and, in parentheses, the total number of servlets. Column *Params* reports the number of injectable parameters in the accessible servlets, with the total number of parameters in parentheses. Non-injectable parameters are state parameters whose purpose is to maintain state, and which are not used to build queries.

5.1.2 Test Input Generation

For each application in the testbed, there are two sets of inputs: *LEGIT*, which consists of legitimate inputs for the application, and

ATTACK, which consists of SQLIAs. The inputs were generated independently by a Master's level student with experience in developing commercial penetration testing tools for Web applications. Test inputs were not generated for non-accessible servlets and for state parameters.

To create the ATTACK set, the student first built a set of potential attack strings by surveying different sources: exploits developed by professional penetration-testing teams to take advantage of SQL-injection vulnerabilities; online vulnerability reports, such as US-CERT (`http://www.us-cert.gov/`) and CERT/CC Advisories (`http://www.cert.org/advisories/`); and information extracted from several security-related mailing lists. The resulting set of attack strings contained 30 unique attacks that had been used against applications similar to the ones in the testbed. All types of attacks reported in the literature [10] were represented in this set except for multi-phase attacks such as overly-descriptive error messages and second-order injections. Since multi-phase attacks require human intervention and interpretation, we omitted them to keep our testbed fully automated. The student then generated a complete set of inputs for each servlet's injectable parameters using values from the set of initial attack strings and legitimate values. The resulting ATTACK set contained a broad range of potential SQLIAs.

The LEGIT set was created in a similar fashion. However, instead of using attack strings to generate sets of parameters, the student used legitimate values. To create "interesting" legitimate values, we asked the student to create inputs that would stress and possibly break naïve SQLIA detection techniques (*e.g.*, techniques based on simple identification of keywords or special characters in the input). The result was a set of legitimate inputs that contained SQL keywords, operators, and troublesome characters, such as single quotes and comment operators.

5.2 Experiment Protocol

To address the first two research questions, we ran the ATTACK and LEGIT input sets against the testbed applications and assessed WASP's effectiveness in stopping attacks without blocking legitimate accesses. *For* **RQ1**, we ran all of the inputs in the ATTACK set and tracked the result of each attack. The results for **RQ1** are summarized in Table 2. The second column reports the total number of attacks in the LEGIT set for each application. The next two columns report the number of attacks that were successful on the original web applications and on the web applications protected by WASP. (Many of the applications performed input validation of some sort and were able to block a subset of the attacks.) *For* **RQ2**, we ran all of the inputs in the LEGIT set and checked how many of these legitimate accesses WASP allowed to execute. The results for this second study are summarized in Table 3. The table shows the number of legitimate accesses WASP allowed to execute (*# Legitimate Accesses*) and the number of accesses blocked by WASP (*False Positives*).

To address **RQ3**, we computed the overhead imposed by WASP on the subjects. To do this, we measured the times required to run all of the inputs in the LEGIT set against instrumented and uninstrumented versions of each application and compared these two times. To avoid problems of imprecision in the timing measurements, we measured the time required to run the entire LEGIT set and then divided it by the number of test inputs to get a per-access average time. Also, to account for possible external factors beyond our control, such as network traffic, we repeated these measurements 100 times for each application and averaged the results. The study was performed on two machines, a client and a server. The client was a Pentium 4, 2.4Ghz, with 1GB memory, running GNU/Linux 2.4. The server was a dual-processor Pentium D, 3.0Ghz, with 2GB of memory, running GNU/Linux 2.6.

Table 2: Results for effectiveness in SQLIAs prevention (RQ1).

Subject	*Total # Attacks*	*Successful Attacks*	
		Original Web Apps	WASP *Protected Web Apps*
Checkers	4,431	922	0
Office Talk	5,888	499	0
Empl. Dir.	6,398	2,066	0
Bookstore	6,154	1,999	0
Events	6,207	2,141	0
Classifieds	5,968	1,973	0
Portal	6,403	3,016	0

Table 3: Results for false positives (RQ2).

Subject	*# Legitimate Accesses*	*False Positives*
Checkers	1,359	0
Office Talk	424	0
Empl. Dir.	658	0
Bookstore	607	0
Events	900	0
Classifieds	574	0
Portal	1,080	0

Table 4 shows the results of this study. For each subject, the table reports the number of inputs in the LEGIT set (*# Inputs*); the average time per database access (*Avg Access Time*); the average time overhead per access (*Avg Overhead*); and the average time overhead as a percentage (*% Overhead*). In the table, all absolute times are expressed in milliseconds.

5.3 Discussion of Results

Overall, the results of our studies indicate that WASP is an effective technique for preventing SQLIAs. In our evaluation, WASP was able to correctly identify all SQLIAs without generating any false positives. In total, WASP stopped 12,616 viable SQLIAs and correctly allowed 5,602 legitimate accesses to the applications.

In most cases, the runtime average imposed by WASP was very low. For the seven applications, the average overhead was 5ms (6%). For most Web applications, this cost is low enough that it would be dominated by the cost of the network and database accesses. One application, Portal, incurred an overhead considerably higher than the other applications (but still negligible in absolute terms). We determined that the higher overhead was due to the fact that Portal generates a very large number of string-based lookup tables. Although these strings are not used to build queries, WASP associates trust markings to them and propagates these markings at runtime. The optimizations discussed in Section 4.2 would eliminate this issue and reduce the overhead considerably.

The main threat to the external validity of our results is that the set of applications and attacks considered in the studies may not be representative of real world applications and attacks. However, all but two of the considered applications are commercial applications, and all have been used in other related studies. Also, to generate our set of attacks, we employed the services of a Master's level student who had experience with SQLIAs, penetration testing, and Web scanners, but was not familiar with our technique. Finally, the attack strings used by the student as a basis for the generation of the attacks were based on real-world SQLIAs.

Table 4: Results for overhead measurements (RQ3).

Subject	*# Inputs*	*Avg Access Time (ms)*	*Avg* Overhead (ms)	*% Overhead*
Checkers	1,359	122	5	5%
Office Talk	424	56	1	2%
Empl. Dir.	658	63	3	5%
Bookstore	607	70	4	6%
Events	900	70	1	1%
Classifieds	574	70	3	5%
Portal	1,080	83	16	19%

6. RELATED WORK

The use of dynamic tainting to prevent SQLIAs has been investigated by several researchers. The two approaches most similar to ours are those by Nguyen-Tuong and colleagues [20] and Pietraszek and Berghe [21]. Similar to them, we track taint information at the character level and use a syntax-aware evaluation to examine tainted input. However, our approach differs from theirs in several important aspects. First, our approach is based on the novel concept of positive tainting, which is an inherently safer way of identifying trusted data (see Section 3.1). Second, we improve on the idea of syntax-aware evaluation by (1) using a database parser to interpret the query string before it is executed, thereby ensuring that our approach can handle attacks based on alternate encodings, and (2) providing a flexible mechanism that allows different trust policies to be associated with different input sources. Finally, a practical advantage of our approach is that it has more lightweight deployment requirements. Their approaches require the use of a customized PHP runtime interpreter, which adversely affects the portability of the approaches.

Other dynamic tainting approaches more loosely related to our approach are those by Haldar, Chandra, and Franz [8] and Martin, Livshits, and Lam [17]. Although they also propose dynamic tainting approaches for Java-based applications, their techniques differ significantly from ours. First, they track taint information at the level of granularity of strings, which introduces imprecision in modeling string operations. Second, they use declassification rules, instead of syntax-aware evaluation, to assess whether a query string contains an attack. Declassification rules assume that sanitizing functions are always effective, which is an unsafe assumption and may leave the application vulnerable to attacks—in many cases, attack strings can pass through sanitizing functions and still be harmful. Another dynamic tainting approach, proposed by Newsome and Song [19], focuses on tainting at a level that is too low to be used for detecting SQLIAs and has a very high execution overhead.

Researchers also proposed dynamic techniques against SQLIAs that do not rely on tainting. These techniques include Intrusion Detection Systems (IDS) and automated penetration testing tools. Scott and Sharp propose Security Gateway [23], which uses developer-provided rules to filter Web traffic, identify attacks, and apply preventive transformations to potentially malicious inputs. The success of this approach depends on the ability of developers to write accurate and meaningful filtering rules. Similarly, Valeur and colleagues [25] developed an IDS that uses machine learning to distinguish legitimate and malicious queries. Their approach, like most learning-based techniques, is limited by the quality of the IDS training set. Machine learning was also used in WAVES [12], an automated penetration testing tool that probes websites for vulnerability to SQLIAs. Like all testing tools, WAVES cannot provide any guarantees of completeness. SQLrand [2] appends a random token to SQL keywords and operators in the application code. A proxy server then checks to make sure that all keywords and operators contain this token before sending the query to the database. Because the SQL keywords and operators injected by an attacker would not contain this token, they would be easily recognized as attacks. The drawbacks of this approach are that the secret token could be guessed, so making the approach ineffective, and that the approach requires the deployment of a special proxy server.

Model-based approaches against SQLIAs include AMNESIA [9], SQL-Check [24], and SQLGuard [3]. AMNESIA, previously developed by two of the authors, combines static analysis and runtime monitoring to detect SQLIAs. The approach uses static analysis to build models of the different types of queries an application can generate and dynamic analysis to intercept and check the query strings generated at runtime against the model. Non-conforming queries are identified as SQLIAs. Problems with this approach are that it is dependent on the precision and efficiency of its underlying static analysis, which may not scale to large applications. Our new technique takes a purely dynamic approach to preventing SQLIAs, thereby eliminating scalability and precision problems. In [24], Su and Wassermann present a formal definition of SQLIAs and propose a sound and complete (under certain assumptions) algorithm that can identify all SQLIAs by using an augmented grammar and by distinguishing untrusted inputs from the rest of the strings by means of a marking mechanism. The main weakness of this approach is that it requires the manual intervention of the developer to identify and annotate untrusted sources of input, which introduces incompleteness problems and may lead to false negatives. Our use of positive tainting eliminates this problem while providing similar guarantees in terms of effectiveness. SQLGuard [3] is an approach similar to SQLCheck. The main difference is that SQLGuard builds its models on the fly by requiring developers to call a special function and to pass to the function the query string before user input is added.

Other approaches against SQLIAs rely purely on static analysis [13, 14, 15, 27]. These approaches scan the application and leverage information flow analysis or heuristics to detect code that could be vulnerable to SQLIAs. Because of the inherently imprecise nature of the static analysis they use, these techniques can generate false positives. Moreover, since they rely on declassification rules to transform untrusted input into safe input, they can also generate false negatives. Wassermann and Su propose a technique [26] that combines static analysis and automated reasoning to detect whether an application can generate queries that contain tautologies. This technique is limited, by definition, in the types of SQLIAs that it can detect.

Finally, researchers have also focused on ways to directly improve the code of an application and eliminate vulnerabilities. Defensive coding best practices [11] have been proposed as a way to eliminate SQL injection vulnerabilities. These coding practices have limited effectiveness because they mostly rely on the ability and training of the developer. Moreover, there are many well-known ways to evade certain types of defensive-coding practices, including "pseudo-remedies" such as stored procedures and prepared statements (*e.g.*, [1, 16, 11]). Researchers have also developed special libraries that can be used to safely create SQL queries [4, 18]. These approaches, although highly effective, require developers to learn new APIs for developing queries, are very expensive to apply on legacy code, and sometimes limit the expressiveness of SQL. Finally, JDBC-Checker [6, 7] is a static analysis tool that detects potential type mismatches in dynamically generated queries. Although it was not intended to prevent SQLIAs, JDBC-Checker can be effective against SQLIAs that leverage vulnerabilities due to type-mismatches, but will not be able to prevent other kinds of SQLIAs.

7. CONCLUSION

We presented a novel, highly automated approach for detecting and preventing SQL injection attacks in Web applications. Our basic 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 become SQL keywords or operators in query strings. Unlike previous approaches based on dynamic tainting, our technique is based on positive tainting, which explicitly identifies trusted (rather than untrusted) data in the program. In this way, we eliminate the problem of false negatives that may result from the incomplete identification of all untrusted data sources. False positives, while 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. The approach 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 several 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 experimental infrastructure are available to other researchers.

We have three immediate goals for future work. The first goal is to further improve the efficiency of the technique. To this end, we will use static analysis to reduce the amount of instrumentation required by the approach. The second goal is to implement the approach for binary applications, by leveraging a binary instrumentation framework and defining a version of the MetaStrings library that works at the binary level. Finally, we plan to evaluate our technique in a completely realistic context, by protecting one of the Web applications running at Georgia Tech with WASP and assessing the effectiveness of WASP in stopping real attacks directed at the application while allowing legitimate accesses.

Acknowledgments

This work was supported by NSF awards CCR-0306372 and CCF-0438871 to Georgia Tech and by the Department of Homeland Security and US Air Force under Contract No. FA8750-05-2-0214. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the US Air Force.

8. REFERENCES

[1] C. Anley. Advanced SQL Injection In SQL Server Applications. White paper, Next Generation Security Software Ltd., 2002.

[2] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proc. of the 2nd Applied Cryptography and Network Security Conf. (ACNS '04)*, pages 292–302, Jun. 2004.

[3] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In *Proc. of the 5th Intl. Workshop on Software Engineering and Middleware (SEM '05)*, pages 106–113, Sep. 2005.

[4] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *Proc. of the 27th Intl. Conference on Software Engineering (ICSE 2005)*, pages 97–106, May 2005.

[5] T. O. Foundation. Top ten most critical web application vulnerabilities, 2005. `http://www.owasp.org/documentation/topten.html`.

[6] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *Proc. of the 26th Intl. Conference on Software Engineering (ICSE 04) – Formal Demos*, pages 697–698, May 2004.

[7] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proc. of the 26th Intl. Conference on Software Engineering (ICSE 04)*, pages 645–654, May 2004.

[8] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proc. of the 21st Annual Computer Security Applications Conference*, pages 303–311, Dec. 2005.

[9] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proc. of the IEEE and ACM Intl. Conference on Automated Software Engineering (ASE 2005)*, pages 174–183, Long Beach, CA, USA, Nov. 2005.

[10] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proc. of the Intl. Symposium on Secure Software Engineering*, Mar. 2006.

[11] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, Washington, Second Edition, 2003.

[12] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proc. of the 12th Intl. World Wide Web Conference (WWW 03)*, pages 148–159, May 2003.

[13] 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.

[14] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *2006 IEEE Symposium on Security and Privacy*, May 2006.

[15] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium*, Aug. 2005.

[16] O. Maor and A. Shulman. SQL Injection Signatures Evasion. White paper, Imperva, Apr. 2004. `http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html`.

[17] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In *OOPSLA '05: Proc. of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 365–383, Oct. 2005.

[18] R. McClure and I. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *Proc. of the 27th Intl. Conference on Software Engineering (ICSE 05)*, pages 88–96, May 2005.

[19] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb. 2005.

[20] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Twentieth IFIP Intl. Information Security Conference (SEC 2005)*, May 2005.

[21] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In *Proc. of Recent Advances in Intrusion Detection (RAID2005)*, Sep. 2005.

[22] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, Sep 1975.

[23] D. Scott and R. Sharp. Abstracting Application-level Web Security. In *Proc. of the 11^{th} Intl. Conference on the World Wide Web (WWW 2002)*, pages 396–407, May 2002.

[24] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *The 33rd Annual Symposium on Principles of Programming Languages*, pages 372–382, Jan. 2006.

[25] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *Proc. of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Vienna, Austria, Jul. 2005.

[26] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In *Proc. of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, pages 70–78, Oct. 2004.

[27] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium*, July 2006.