# WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation

William G.J. Halfond, Alessandro Orso, *Member*, *IEEE Computer Society*, and
Panagiotis Manolios, *Member*, *IEEE Computer Society*

**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. 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 incurs a negligible 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 empirical evaluation on a wide range of Web applications that we subjected to a large and varied set of attacks and legitimate accesses. WASP was able to stop all of the otherwise successful attacks and did not generate any false positives.

**Index Terms**—Security, SQL injection, dynamic tainting, runtime monitoring.

## 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 [6]. 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.

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 intent is that Web applications will limit the kinds of queries that can be generated to a safe subset of all possible queries, regardless of what input users provide. However, inadequate 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 disastrous and can range from leaking of sensitive data (for example, customer data) to the destruction of database contents.

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. For example, one common class of solutions is based on defensive coding practices, which have been less than successful for three main 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. Third, legacy software poses a particularly difficult problem because of the cost and complexity of retrofitting existing code so that it is compliant with defensive coding practices.

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.

The kind of dynamic tainting that we use gives our approach several important advantages over techniques based on other mechanisms. Many techniques rely on complex static analyses in order to find potential vulnerabilities in the code (for example, [11], [18], [29]). These kinds of conservative static analyses can generate high rates of false positives and can have scalability issues when

- *W.G.J. Halfond and A. Orso are with the College of Computing, Georgia Institute of Technology, Klaus Advanced Computing Building, 266 Ferst Drive, Atlanta, GA 30332-0765. E-mail: {whalfond, orso}@cc.gatech.edu.*
- *P. Manolios is with the College of Computer and Information Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115. E-mail: pete@ccs.neu.edu.*

*Manuscript received 24 Feb. 2007; revised 8 Aug. 2007; accepted 29 Aug. 2007; published online 24 Sept. 2007.*
*Recommended for acceptance by P. McDaniel and B. Nuseibeh.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0084-0207.*
*Digital Object Identifier no. 10.1109/TSE.2007.70748.*

0098-5589/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

applied to large complex applications. In contrast, our approach does not rely on complex static analyses and is both efficient and precise. Other techniques involve extensive human effort (for example, [5], [21], [27]). They require developers to manually rewrite parts of the Web applications, build queries using special libraries, or mark all points in the code at which malicious input could be introduced. Our approach is highly automated and, in most cases, requires minimal or no developer intervention. Last, several proposed techniques require the deployment of extensive infrastructure or involve complex configurations (for example, [2], [26], [28]). Our approach does not require additional infrastructure and can be automatically deployed.

Compared to other existing techniques based on dynamic tainting (for example, [9], [23], [24]), 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, sources of trusted data can more easily and accurately be identified than untrusted data sources. Therefore, the use of positive tainting leads to increased automation. Second, the two approaches significantly differ in how they are affected by incompleteness. With negative tainting, failure to identify the complete set of untrusted data sources can result in false negatives, that is, successful and undetected attacks. With positive tainting, missing trusted data sources can result in false positives (that is, legitimate accesses can be prevented from completing). False positives that occur in the field would be problematic. Using our approach, however, false positives are likely to be detected during prerelease testing. Our approach provides specific mechanisms for helping developers detect false positives early, identify their sources, and easily eliminate them in future runs by tagging the identified sources as trusted.

The *second conceptual advantage* of our approach is the use of flexible syntax-aware evaluation. Syntax-aware evaluation lets us address security problems that are derived from mixing data and code while still allowing for this mixing to occur. More precisely, it gives developers a mechanism for regulating the usage of string data based not only on its source but also on its syntactical role in a query string. 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 it 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 it must be 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 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 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 listed as follows:

1. a new automated technique for preventing SQLIAs based on the novel concept of positive tainting and on flexible syntax-aware evaluation,
2. a mechanism to perform efficient dynamic tainting of Java strings that precisely propagates trust markings while strings are manipulated at runtime,
3. a tool that implements our SQLIA prevention technique for Java-based Web applications and has minimal deployment requirements, and
4. 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. 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 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. Interested readers can refer to the work of Su and Wassermann [27] for a formal definition of SQLIAs. SQLIAs leverage a wide range of mechanisms and input channels to inject malicious commands into a vulnerable application [12]. 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

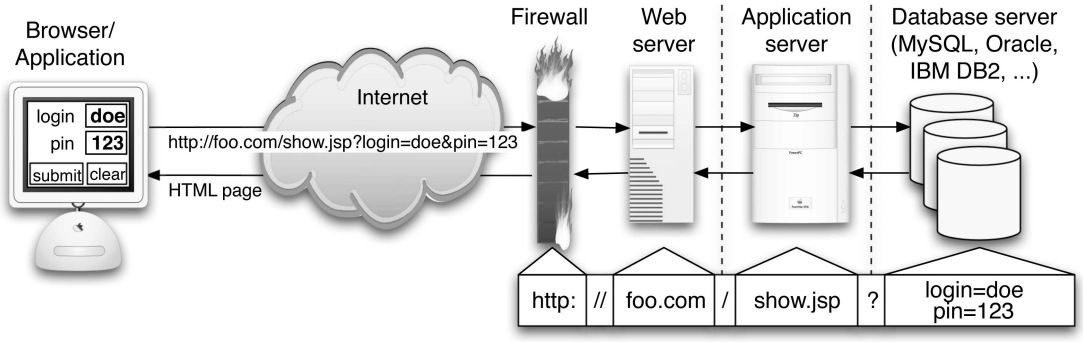


Fig. 1. Example of interaction between a user and a typical Web application.

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.

```
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.

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.

### 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. These techniques go beyond the well-known SQLIA examples and take advantage of esoteric and advanced SQL constructs. Ignoring the existence of these kinds of attacks leads to the development of solutions that only partially address the SQLIA problem.

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. In addition, *second-order injections* use advanced knowledge of vulnerable applications to introduce attacks by using otherwise properly secured input sources [1]. A developer may suitably escape, type-check, and filter input that comes from the user and assume that it is safe. Later on, when that data is used in a different context or to build a different type of query, the previously safe input may enable an injection attack.

Once attackers have identified an input source that can be used to exploit an SQLIA vulnerability, there are many different types of attack techniques that they can leverage. 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

summarize the main techniques for performing SQLIAs. We provide additional information and examples of how these techniques work in [12].

#### 2.1.1 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.

#### 2.1.2 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 "' `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='' UNION SELECT cardNo
from CreditCards where acctNo=7032 -- AND pin=
```

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.

#### 2.1.4 Malformed Queries

Union queries and piggybacked queries let attackers perform specific queries or execute specific commands on a database, but require some prior knowledge of the database schema, which is often unknown. Malformed queries allow for overcoming this problem by taking advantage of overly descriptive error messages that are generated by the database when a malformed query is rejected. When these messages are directly returned to the user of the Web application, instead of being logged for debugging by developers, attackers can make use of the debugging information to identify vulnerable parameters and infer the schema of the underlying database. Attackers exploit this situation by injecting SQL tokens or garbage input that causes the query to contain syntax errors, type mismatches, or logical errors. Considering our example, an attacker could try causing a type mismatch error by injecting the following text into the *pin* input field: "`convert(int, (select top 1 name from sysobjects where xtype` $=$ `'u'))`." The resulting query generated by the Web application is the following:

```
SELECT acct FROM users WHERE login='' AND pin=convert(int,
(select top 1 name from sysobjects where xtype='u'))
```

The injected query extracts the name of the first user table `xtype` $=$ `'u'` from the database's metadata table `sysobjects`. It then converts this table name to an integer. Because the name of the table is a string, the conversion is illegal and the database returns an error. For example, a SQL Server may return the following error: "*Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int.*" From this message, the attacker can 1) see that the database is an SQL Server and 2) discover that the name of the first user-defined table in the database is "CreditCards" (the string that caused the type conversion to occur). A similar strategy can be used to systematically extract the name and type of each column in the given table. Using this information about the schema of the database, an attacker can create

more precise attacks that specifically target certain types of information. Malformed queries are typically used as a preliminary information-gathering step for other attacks.

#### 2.1.5 Inference

Similar to malformed queries, inference-based attacks let attackers discover information about a database schema. This type of SQLIAs creates queries that cause an application or database to behave differently based on the results of the query. This way, even if an application does not directly provide the results of the query to the attacker, it is possible to observe side effects caused by the query and deduce its results. One particular type of attack based on inference is a *timing attack*, which lets attackers gather information from a database by observing timing delays in the database's responses. To perform a timing attack, attackers structure their injected queries in the form of an if-then statement whose branch condition corresponds to a question about the contents of the database. The attacker then uses the `WAITFOR` keyword along one of the branches, which causes the database to delay its response by a specified time. By measuring the increase or decrease in the database response time, attackers can infer which branch was taken and the answer to the injected question. For our example servlet, an attacker could inject the following text into the *login* parameter: "`legalUser' AND ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 --`." This injection produces the following query:

```
SELECT acct FROM users WHERE login='legalUser' and
ASCII(SUBSTRING((select top 1 name from sysobjects),1,1))
> X WAITFOR 10 -- ' AND pin=
```

In the attack, the `SUBSTRING` function is used to extract the first character of the database's first table's name, which is then converted into an ASCII value and compared with the value of $X$. If the value is greater, the attacker will be able to observe a 10 s delay in the database response. The attacker can continue this way and use a binary-search strategy to identify the value of each character in the table's name. Another well-known type of inference attack is *blind* SQL injection [12].

#### 2.1.6 Alternate Encodings

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." Even if developers account for alternate encodings, this technique can still be successful because alternate encodings can target different layers in the application. For example, a developer may scan for a Unicode or hexadecimal encoding of a single quote and not realize that the attacker can leverage database functions to encode the same character. An effective code-based defense against alternate encodings requires developers to be aware of all of the possible encodings that could affect a given query string as it passes through the different application layers. Because developing such a complete protection is very difficult in practice, attackers have been successful in using alternate encodings to conceal attack strings. The following example attack (from [13]) shows the level of obfuscation that can be achieved using alternate encodings. In the attack, the *pin* field is injected with string "`0; exec(char(0x73687574646f776e))`," which results in the following query:

```
SELECT acct FROM users WHERE login='' AND pin=0;
exec(char(0x73687574646f776e))
```

This attack leverages the `char()` function provided by some databases and uses ASCII hexadecimal encoding. The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the attack string. This encoded string is inserted into a query by using some other type of attack profile and, when it is executed by the database, translates into the `shutdown` command.

#### 2.1.7 Leveraging Stored Procedures

Another strongly advertised solution for the problem of SQLIAs is the use of stored procedures, that is, procedures that are stored in the database and can be run by the database engine. Stored procedures provide developers with an extra layer of abstraction because they can enforce businesswide database rules, independent of the logic of individual Web applications. Unfortunately, it is a common misconception that the mere use of stored procedures protects an application from SQLIAs: Similarly to any other software, the safety of stored procedures depends on the way in which they are coded and on the use of adequate defensive coding practices. Therefore, parametric stored procedures could also be vulnerable to SQLIAs, just like the rest of the code in a Web application.

The following example demonstrates how a (parametric) stored procedure can be exploited via an SQLIA. In this scenario, assume that the query string constructed by our example servlet has been replaced by a call to the following stored procedure:

```
CREATE PROCEDURE DBO.isAuthenticated
    @userName varchar2, @pin int
AS
    EXEC("SELECT acct FROM users WHERE login='"
    + @userName + "' and pin= " +@pin);
GO
```

To perform an SQLIA that exploits this stored procedure, the attacker can simply inject the text "`' ; SHUTDOWN; --`" into the `userName` field. This injection causes the stored procedure to generate the following query, which would result in the database being shut down:

```
SELECT acct FROM users WHERE login=' ';SHUTDOWN; -- AND pin=
```

## 3 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.

### 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 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 [25]: 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

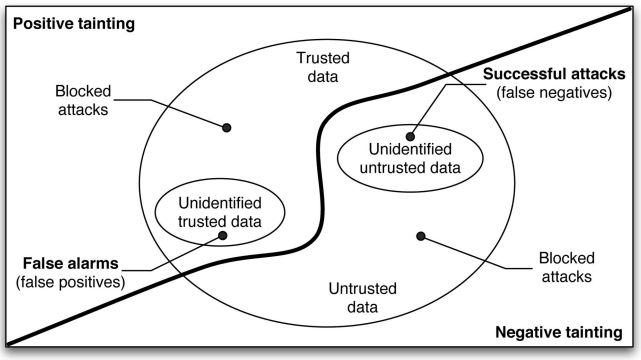


Fig. 3. Identification of trusted and untrusted data.

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.[1] This is because 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 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 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 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. Our approach

1. Without loss of generality, we assume that developers are trustworthy. An attack encoded in the application by a developer would not be an SQLIA but a form of backdoor, which is not the problem addressed in this work.

uses this information to mark data that comes from these additional sources as trusted.

In a typical scenario, we expect developers to specify most of the trusted sources before testing and deployment. However, some of these sources might be overlooked until after a false positive is reported, in which case, developers would add the omitted items to the list of trusted sources. In this process, the set of trusted data sources monotonically grows 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 applications. Therefore, even when developers fail to completely identify additional sources of trusted data beforehand, we expect these sources to be identified during normal testing and the set of trusted data to quickly converge to the complete set.

It is also worth noting that none of the subjects that we collected and examined so far required us to specify additional trusted data sources. All of these subjects used only hard-coded strings to build query strings.

### 3.2 Accurate and Efficient 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 the "right" 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. Another alternative would be to trace taint data at the bit level, which would allow us to account for situations where string data are manipulated as character values using bitwise operators. However, operating at the bit level would make the approach considerably more expensive and complex to implement and deploy. Most importantly, our experience with Web applications shows that working at a finer level of granularity than a character would not yield any benefit in terms of effectiveness. Strings are typically manipulated using methods provided by string library classes and we have not encountered any case of query strings that are manipulated at the bit level.

*Accounting for string manipulations.* To accurately maintain character-level taint information, we must identify all relevant string operations and account for their effect on the taint markings (that is, we must enforce complete mediation of all string operations). Our approach achieves this goal by taking advantage of the encapsulation offered by object-oriented languages, in particular by Java, in which all string manipulations are performed using a small set of classes and methods. Our approach extends all such classes and methods by adding functionality to update taint markings based on the methods' semantics.

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

### 3.3 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 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 (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 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.

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] (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 $(\texttt{id}|\texttt{severity}) =' \backslash \texttt{w} +' \ ((\texttt{AND}|\texttt{OR}) \ (\texttt{id}|\texttt{severity}) =' \backslash \texttt{w}+')\star$.

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.

### 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). In contrast, our approach has minimal deployment requirements and is efficient, which makes it practical for use in real settings. Our technique does not necessitate a customized runtime system. It requires only minor localized instrumentation of the application to 1) enable the use of our string library and 2) insert the calls that perform syntax-aware evaluation of a query before it is sent to the database. The protected application is then deployed as a normal Web application except that the deployment must include our string library. Both instrumentation and deployment are fully automated. We discuss the deployment requirements and the overhead of the approach in greater detail in Sections 4.5 and 5.3.

## 4 Our Implementation: WASP

To evaluate our approach, we developed a prototype tool called WASP (Web Application SQL-injection Preventer), which is written in 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. (We discuss the applicability of the approach in other contexts in Section 4.1.)

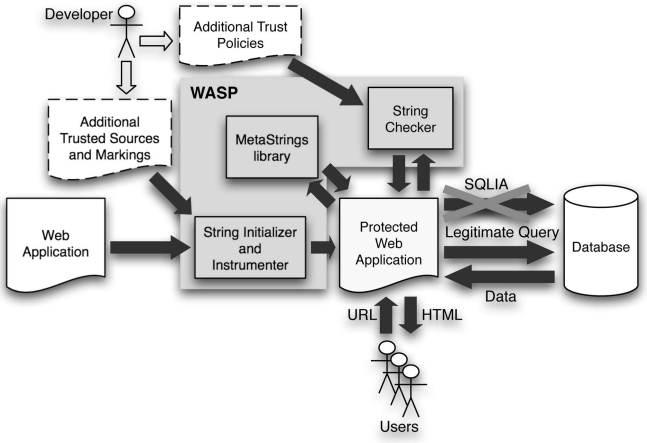


Fig. 4. High-level overview of the approach and tool.

Fig. 4 shows the high-level architecture of WASP. As this 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 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 `StringBuffer`).[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 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 relevant methods of the original class and 2) replacing instantiations of the original class with instantiations of the subclass. In our implementation, we leverage the object-oriented features of Java, and the

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

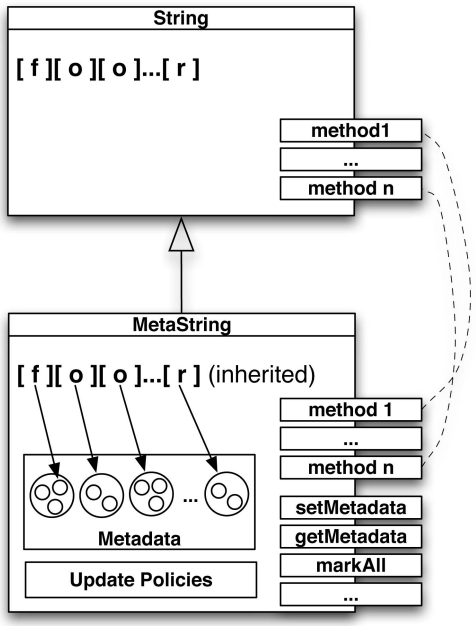


Fig. 5. An intuitive view of a MetaStrings library class.

approach should be easily applicable to applications built using other object-oriented languages such as .NET. Although the use of object-oriented features allows our current implementation to be elegant and minimally intrusive, we expect the approach to be portable, with suitable engineering, to non-object-oriented languages. For example, in C, the approach could be implemented by identifying and instrumenting calls to functions and operations that manipulate strings or characters. In general, our approach should be portable to all contexts where 1) string-creation and string-manipulation operations can be identified and 2) a character-level taint initialization and propagation mechanism can be implemented (either through instrumentation or by modifying the runtime system).

To illustrate our MetaStrings library with an example, Fig. 5 shows an intuitive view of the MetaStrings class that corresponds to Java's `String` class. As this 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 completely defined at the application level and thus does not require a customized runtime system.
4. Its usage requires only minimal and automatically performed changes in the application's bytecode.
5. It imposes a low execution overhead on Web applications, as shown in 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 markings 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 metaclasses. 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 (for example, representing strings by using primitive `char` values). Moreover, during the 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 WASP must consider: hard-coded strings, strings implicitly created by Java, and strings originating in external sources. In the following sections, we explain how strings from each category are identified and marked.

#### *4.2.1 Hard-Coded Strings*

The identification of hard-coded strings in an application's bytecode is fairly straightforward. In Java, hard-coded strings are represented using `String` objects that are automatically created 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 by 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.

Fig. 6 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 Fig. 2), which creates one of the hard-coded strings in the servlet. Underneath, we show the original bytecode (left column) and the modified bytecode (right column). The modified bytecode contains additional instructions that 1) load a new `MetaString` object on the

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

Fig. 6. Instrumentation for hard-coded strings.

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

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

Fig. 7. Instrumentation for implicitly created strings.

stack, 2) call the `MetaString` constructor by 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.

#### 4.2.2 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 situation, 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.

Fig. 7 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.

#### 4.2.3 Strings from External Sources

To use query fragments that come 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 that come from a file. In the configuration file, the developer specifies the name of the file (for example, `foo.txt`) as a trusted source of strings. Based on this information, WASP scans the bytecode for all instantiations of new file objects (that is, `File`, `FileInputStream`, and `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 the `Buffered`

Reader'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 filenames at runtime, we match hostnames and ports. The interaction with databases is more complicated and requires WASP to not only match the initiating connection but also 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 unnecessary instrumentation. We could reduce 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 with the construction of query strings and therefore do not need to be instrumented. A static analysis could also identify cases where the filename associated with a file object is never one of the developer-specified trusted filenames and avoid instrumenting that object and subsequent operations on it. Analogous optimizations could be implemented for other external sources. We did not incorporate any of these optimizations in the current tool because the overhead imposed by our current (conservative) implementation was insignificant in most of the cases.

### 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 the "learning mode," that would typically be used during in-house testing. When in the 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 the 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 a false positive, knowing the position in the code of the offending string(s) helps developers in correcting 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*, that is, 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 conservatively 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 as a parameter the MetaStrings object that contains the query about to be executed.

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 by using a 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] (see Section 2.1.6). Our current implementation includes parsers for SQL-92 (ANSI) and PostgreSQL and allows for adding other parsers in a modular fashion. 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, an attack is identified. When MetaChecker identifies an attack, it can execute any developer-specified action. In our evaluation, we configured WASP so that it blocked the malicious query from executing and logged the attempted attack.

If developers specify 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, check 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 identifies the query as an SQLIA.

We illustrate how the default policy for syntax-aware evaluation works by 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. Fig. 8 shows the sequence of tokens in the legitimate query as they would be parsed by MetaChecker. In this 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 in trusted strings, MetaChecker simply checks whether all of these tokens are comprised of trusted characters. The query in Fig. 8 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. Fig. 9 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 identify the query as an SQLIA.

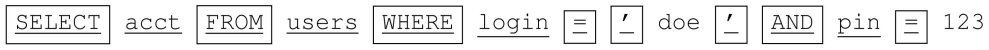


Fig. 8. Example query 1 after parsing by the runtime monitor.

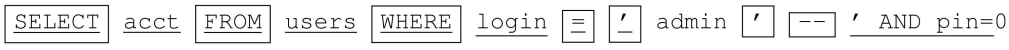


Fig. 9. Example query 2 after parsing by the runtime monitor.

### 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: offline and online. Offline instrumentation statically instruments the application and deploys the instrumented version of the application. Online instrumentation deploys an unmodified application and instruments the code at load time (that is, 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 offline instrumentation and insert 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. (Moreover, this modification is a temporary workaround for the current limitations of Java's instrumentation package.) No modification of the JVM is required. The instrumentation of a Web application is also automatically performed. 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.

## 5 Empirical Evaluation

In our evaluation, we assessed the effectiveness and efficiency of our approach. To do this, we used WASP to protect several real vulnerable Web applications while subjecting them to a large number of attacks and legitimate accesses and investigated 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 are incorrectly identified by WASP as attacks?
- **RQ3.** What is the runtime overhead imposed by WASP on the Web applications that it protects?

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

### 5.1 Experiment Setup

The framework that we use for our experiments consists of a set of vulnerable Web applications, a large set of test inputs that contain both legitimate accesses and SQLIAs, and monitoring and logging tools. We developed the initial framework in our previous work [11] and it has since been used both by us and by other researchers [10], [27]. In this study, we have expanded the framework by 1) including additional open source Web applications with known vulnerabilities, 2) generating legitimate and malicious inputs for these new applications, and 3) expanding the set of inputs for the existing applications. In the next two sections, we discuss the Web applications and the set of inputs used in our experiments in more detail.

#### 5.1.1 Software Subjects

Our set of software subjects consists of 10 Web applications that are known to be vulnerable to SQLIAs. Five of the applications are commercial applications that we obtained from GotoCode (http://www.gotocode.com/): Employee Directory, Bookstore, Events, Classifieds, and Portal. Two applications, OfficeTalk and Checkers, are student-developed applications that have been used in related work [8]. Two other applications, Daffodil and Filelister, are open source applications that have been identified in the Open Source Vulnerability Database (http://osvdb.org/, entries 22879 and 21416) as containing one or more SQL injection vulnerabilities. The last subject, WebGoat, is a purposely insecure Web application that was developed by the Open Web Application Security Project (http://www.owasp.org/) to demonstrate common Web application vulnerabilities. Among these 10 subjects, the first seven applications contain a wide range of vulnerabilities, whereas the last three contain specific and known SQLIA vulnerabilities.

Table 1 provides summary information about each of the subjects in our evaluation. It shows, for each subject, its size (*LOC*), number of database interaction points (*DBIs*), number of vulnerable servlets (*Vuln Servlets*), and total number of servlets (*Total Servlets*). We considered all of the servlets in the first seven subjects that accepted user input to be potentially vulnerable because we had no initial information about their vulnerabilities. For the remaining three applications, we considered as vulnerable only those servlets with specific and known vulnerabilities.

#### 5.1.2 Malicious and Legitimate Inputs

For each of the Web applications considered, we created two sets of inputs: *LEGIT*, which consists of legitimate inputs for the application, and *ATTACK*, which consists of SQLIAs.

TABLE 1
Subject Programs for the Empirical Study

| *Subject* | *LOC* | *DBIs* | *Servlets* | |
|---|---|---|---|---|
| | | | *Vuln* | *Total* |
| Checkers | 5,415 | 5 | 33 | 33 |
| Office Talk | 4,670 | 40 | 38 | 38 |
| Employee Directory | 5,529 | 23 | 8 | 11 |
| Bookstore | 19,402 | 71 | 25 | 28 |
| Events | 7,164 | 31 | 11 | 13 |
| Classifieds | 10,702 | 34 | 16 | 19 |
| Portal | 16,089 | 67 | 25 | 28 |
| Daffodil | 18,706 | 156 | 1 | 69 |
| Filelister | 8,671 | 10 | 1 | 10 |
| WebGoat | 20,725 | 184 | 4 | 27 |

TABLE 2
Results of Testing for False Negatives (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 |
| Daffodil | 19 | 19 | 0 |
| Filelister | 96 | 80 | 0 |
| WebGoat | 96 | 88 | 0 |

To create the *ATTACK* sets, we employed a Master's level student with experience in developing commercial penetration testing tools. The student first assembled a list of actual SQLIAs 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 24 unique attacks. All types of attacks reported in the literature [12] were represented in this set, except for multiphase attacks such as second-order injections. Since multiphase attacks require human intervention and interpretation, we omitted them to keep our testbed fully automated. These attack strings were then used to build inputs for all of the vulnerable servlets in each application. The resulting ATTACK sets contained a broad range of potential SQLIAs.

The LEGIT sets were created in a similar fashion. However, instead of using attack strings to generate sets of inputs, the student used legitimate values. To create "interesting" legitimate values, we asked the student to generate input strings that, although legal, would stress and possibly break naive SQLIA detection techniques (for example, techniques based on simple identification of keywords or special characters in the input). For instance, the legitimate values contained SQL keywords (for example, "`SELECT`" and "`DROP`"), query fragments (for example, "`or 1 = 1`"), and properly escaped SQL operators (for example, the single quote `''` and the percent sign `\%`). These values were used to build inputs for the vulnerable servlets that looked "suspicious" without actually resulting in an SQLIA.

## 5.2 Experimental Protocol

**RQ1** addresses the issue of false negatives. To investigate this question, we 1) ran the inputs in the ATTACK sets against our subject applications and 2) tracked the result of each attack to check whether it was detected and prevented by WASP. The results of this evaluation are shown in Table 2. The second column reports the total number of attacks in the application's *ATTACK* set. The next two columns show the number of attacks that were successful against the original unprotected Web application and the number of attacks that were successful on the application protected using WASP. The reason that some attacks were not successful on the unprotected applications is twofold. First, not all of the 24 attack strings represented viable attacks against all vulnerable servlets. Second, many of the applications performed some type of input validation that could catch and prevent a subset of the attempted attacks.

**RQ2** deals with false positives. To address this question, we ran all of the test inputs in each application's *LEGIT* set against the application. As before, we tracked the result of each of these legitimate accesses to see if WASP reported it as an attack, which would be a false positive. The results for this evaluation are summarized in Table 3. The table shows the number of legitimate accesses that WASP allows to execute (# *Legitimate Accesses*) and the number of accesses blocked by WASP (*False Positives*).

To address **RQ3**, we measured the overhead incurred by applications that were protected using WASP. To do this, we measured and compared the times needed to run the LEGIT set against a protected version and an unprotected version of each application. We used only the LEGIT set for this part of the study because our current implementation of WASP terminates the execution when it detects an attack, which would have made the total execution time for the WASP-protected version faster than the time for the normal version. To reduce problems with the precision of the timing measurements, we measured the total time that it

TABLE 3
Results of Testing for False Positives (RQ2)

| *Subject* | *# Legitimate Accesses* | *False Positives* |
|---|---|---|
| Checkers | 1,359 | 0 |
| Office Talk | 424 | 0 |
| Empl. Dir. | 1,244 | 0 |
| Bookstore | 3,239 | 0 |
| Events | 1,324 | 0 |
| Classifieds | 2042 | 0 |
| Portal | 3,435 | 0 |
| Daffodil | 19 | 0 |
| Filelister | 40 | 0 |
| WebGoat | 40 | 0 |

TABLE 4
Overhead Measurements for the Macro Benchmarks (RQ3)

| *Subject* | *# Inputs* | *Avg Time Uninst (ms)* | *Avg Ovhd (ms)* | *% Ovhd* |
|---|---|---|---|---|
| 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% |
| Daffodil | 19 | 90 | 6 | 6% |
| Filelister | 40 | 172 | 1 | 1% |
| WebGoat | 40 | 940 | 40 | 5% |

took to run the entire LEGIT set against an application, instead of the single times for each input in the set, and divided this time by the number of accesses to get an average value. In addition, to account for possible external factors beyond our control, such as network traffic or other OS activities, we repeated these measurements 100 times for each application and averaged the results. All measurements were performed on two machines that act as client and server. The client was a 2.4 GHz Pentium 4 with 1 Gbyte memory, running GNU/Linux 2.4. The server was a 3.0 GHz dual-processor Pentium D with 2 Gbyte memory, running GNU/Linux 2.6.

In addition to measuring the overhead for these macro benchmarks, we also measured the overhead imposed by individual MetaStrings methods on a set of micro benchmarks. To do this, we first identified the methods most commonly used in the subject Web applications, seven methods overall. We then measured, for each method $m$, the runtime of 1) a driver that performed 10,000 calls to the original $m$ and 2) a driver that performed the same number of calls to the MetaStrings version of $m$. As before, we performed the measurements 100 times and averaged the results. This second set of measurements was also performed on a 3.0 GHz dual-processor Pentium D with 2 Gbyte memory, running GNU/Linux 2.6.

Table 4 shows the results of the timing measurements for the macro benchmarks. For each subject, the table reports the number of inputs in the LEGIT set (*# Inputs*), the average time per access for the uninstrumented Web application (*Avg Time Uninst*), the average time overhead per access for the instrumented version (*Avg Overhead*), and the average time overhead as a percentage (*% Overhead*). In the table, all absolute times are expressed in milliseconds. Fig. 10 provides another view of the timing measurements by using a bar chart. In this figure, the total servlet access times for the instrumented and uninstrumented versions are shown side by side. As the figure shows, the difference between the two bars for a subject, which represents the WASP overhead, is small in both relative and absolute terms.

Table 5 reports the results of the timing measurements for the micro benchmarks. For each of the seven *methods* considered, the table shows the average runtime, in milliseconds, for 10,000 executions of the *original method* and of its *MetaStrings version.* Note that the measured overhead is due to either the creation and initialization of a new `Set` object for each character (for the default constructors) or the copying of the trust markings from one object to another (for the parameterized constructors and for the methods `append` and `concat`). Although the measured overhead is considerable in relative terms, it is mostly negligible in absolute terms. In the worst case, for method `StringBuilder.Append(StringBuilder)`, the MetaStrings version of the method takes 71 ms more than its original version for 10,000 executions.

### 5.3 Discussion of the Experimental Results

The results of our evaluation show that, overall, WASP is an effective technique for preventing SQLIAs. In our

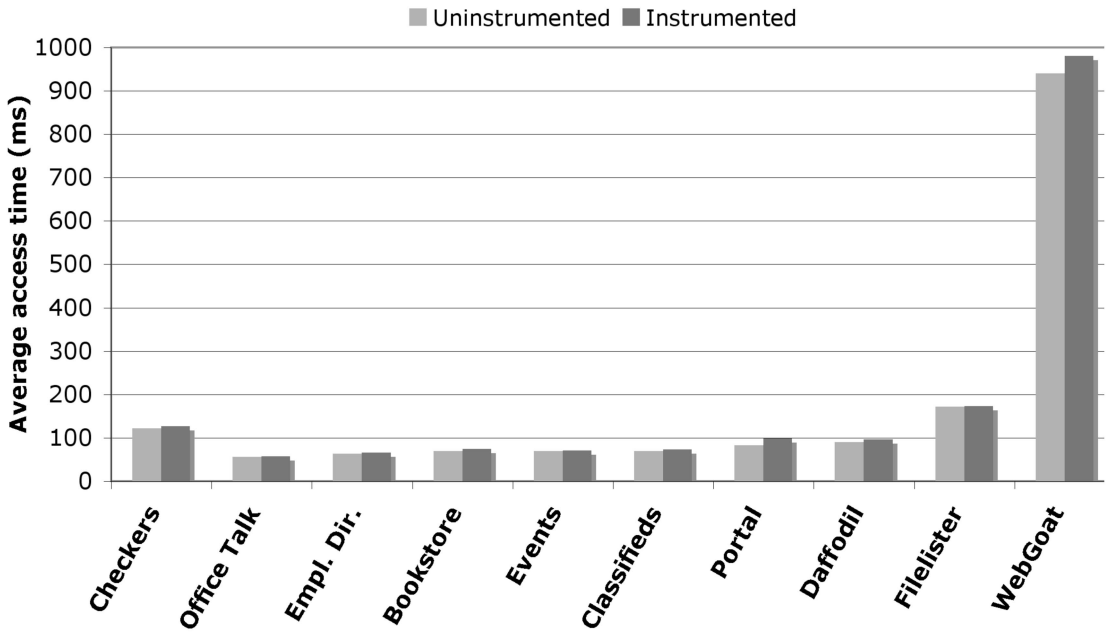


Fig. 10. Runtime overhead imposed by WASP's instrumentation on the subject Web applications.

TABLE 5
Overhead Measurements for the Micro Benchmarks (RQ3)

| *Method* | *Original method* | *MetaStrings version* |
|---|---|---|
| String.Concat(String) | 24 | 65 |
| String.New() | 4 | 9 |
| String.New(String) | 3 | 22 |
| StringBuilder.Append(StringBuilder) | 1 | 72 |
| StringBuilder.Append(String) | 2 | 52 |
| StringBuilder.New() | 2 | 21 |
| StringBuilder.New(String) | 2 | 16 |

evaluation, WASP was able to correctly identify all SQLIAs while generating no false positives. In total, WASP stopped 12,826 viable SQLIAs without preventing any of the 13,166 legitimate accesses from executing.

The overhead imposed by WASP was also relatively low. For the 10 applications, the average overhead was about 8 ms (5.5 percent). For most Web applications, this cost is low enough that it would be dominated by the cost of the network and database accesses. Furthermore, we believe that, by using some of the optimizations discussed in Section 4.2.3, it would be possible to lower this number even further, if deemed necessary, after performing more experimentation. Portal, the application that incurred the highest overhead, is an example of an application that would benefit enormously from these optimizations. Portal generates a large number of string-based lookup tables. Although these strings are not used to build queries, WASP associates trust markings with them and propagates these markings at runtime. In this specific case, a simple dependency analysis would be able to determine that these markings are unnecessary and avoid the overhead associated with these operations.

Like all empirical studies, our evaluation has limitations that may affect the external and internal validity of its results. The primary threat to the external validity of the results is that the attacks and applications used in our studies may not be representative of real-world applications and attacks. To mitigate this issue, we have included in our set of subjects Web applications that come from a number of different sources and were developed using different approaches (for example, the five GotoCode applications are developed using an approach that is based on automated code generation). In addition, our set of attacks was independently developed by a Master's level student who had considerable experience with SQLIAs and penetration testing 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.

For this study, threats to internal validity mainly concern errors in our implementation or in our measurement tools that could affect outcomes. To control these threats, we validated the implementations and tools on small-scale examples and performed a considerable amount of spot checking for some of the individual results.

## 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 et al. [23] and Pietraszek and Berghe [24]. Similarly, 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 et al. [9] and Martin et al. [20]. Although they also propose dynamic tainting approaches for Java-based applications, their techniques significantly differ 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 may still be harmful. Another dynamic tainting approach, proposed by Newsome and Song [22], focuses on tainting at a level that is too low to be used for detecting SQLIAs and has a very high execution overhead. Xu et al. [31] propose a generalized tainting mechanism that can address a wide range of input-validation-related attacks, targets C programs, and works by instrumenting the code at the source level. Their approach can be considered a framework for performing dynamic taint analysis on C programs. As such, it could be leveraged to implement a version of our approach for C-based Web applications.

Researchers also proposed dynamic techniques against SQLIAs that do not rely on tainting. These techniques include Intrusion Detection Systems (IDSs) and automated penetration testing tools. Scott and Sharp propose Security Gateway [26], 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 et al. [28] 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 [14], an automated penetration testing tool that probes Web sites 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, thus making the approach ineffective, and that the approach requires the deployment of a special proxy server.

Model-based approaches against SQLIAs include AMNESIA [11], SQL-Check [27], 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 that an application can generate and dynamic analysis to intercept and check the query strings generated at runtime against the model. Queries that do not match the model are identified as SQLIAs. A problem with this approach is 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. SQLCheck [27] identifies SQLIAs by using an augmented grammar and 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 [15], [16], [17], [18], [30]. 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 that 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 [29] 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 investigated ways to statically eliminate vulnerabilities from the code of a Web application. Defensive coding best practices [13] have been proposed as a possible approach, but they have limited effectiveness because they rely almost exclusively on the ability and training of developers. Moreover, there are many well-known ways to evade some defensive-coding practices, including "pseudoremedies" such as stored procedures and prepared statements (for example, [1], [13], [19]). Researchers have also developed special libraries that can be used to safely create SQL queries [5], [21]. These approaches, although highly effective, require developers to learn new APIs, can be very expensive to apply on legacy code, and sometimes limit the expressiveness of SQL. Finally, JDBC-Checker [7], [8] 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

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.

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 [4].

## Acknowledgments

This work was supported by US National Science Foundation Awards CCF-0438871 and CCF-0541080 to Georgia Tech and by the US Department of Homeland Security and US Air Force under Contract 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. The anonymous reviewers provided useful feedback that helped improve the quality of this paper.

## References

[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] G.T. Buehrer, B.W. Weide, and P.A.G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," *Proc. Fifth Int'l Workshop Software Eng. and Middleware,* pp. 106-113, Sept. 2005.

[4] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 196-206, July 2007.

[5] 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.

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

[7] C. Gould, Z. Su, and P. Devanbu, "JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications," *Proc. 26th Int'l Conf. Software Eng.,* formal demos, pp. 697-698, May 2004.

[8] C. Gould, Z. Su, and P. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," *Proc. 26th Int'l Conf. Software Eng.,* pp. 645-654, May 2004.

[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.

[10] W. Halfond, A. Orso, and P. Manolios, "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks," *Proc. ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 175-185, Nov. 2006.

[11] W.G. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," *Proc. 20th IEEE and ACM Int'l Conf. Automated Software Eng.,* pp. 174-183, Nov. 2005.

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

[13] M. Howard and D. LeBlanc, *Writing Secure Code,* second ed. Microsoft Press, 2003.

[14] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," *Proc. 12th Int'l Conf. World Wide Web,* pp. 148-159, May 2003.

[15] 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.

[16] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," *Proc. IEEE Symp. Security and Privacy,* May 2006.

[17] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise Alias Analysis for Static Detection of Web Application Vulnerabilities," *Proc. Workshop Programming Languages and Analysis for Security,* pp. 27-36, June 2006.

[18] V.B. Livshits and M.S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," *Proc. 14th Usenix Security Symp.,* Aug. 2005.

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

[20] M. Martin, B. Livshits, and M.S. Lam, "Finding Application Errors and Security Flaws Using PQL: A Program Query Language," *Proc. 20th Ann. ACM SIGPLAN Conf. Object Oriented Programming Systems Languages and Applications,* pp. 365-383, Oct. 2005.

[21] R. McClure and I. Krüger, "SQL DOM: Compile Time Checking of Dynamic SQL Statements," *Proc. 27th Int'l Conf. Software Eng.,* pp. 88-96, May 2005.

[22] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proc. 12th Ann. Network and Distributed System Security Symp.,* Feb. 2005.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically Hardening Web Applications Using Precise Tainting Information," *Proc. 20th IFIP Int'l Information Security Conf.,* May 2005.

[24] T. Pietraszek and C.V. Berghe, "Defending against Injection Attacks through Context-Sensitive String Evaluation," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection,* Sept. 2005.

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

[26] D. Scott and R. Sharp, "Abstracting Application-Level Web Security," *Proc. 11th Int'l Conf. World Wide Web,* pp. 396-407, May 2002.

[27] 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.

[28] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks," *Proc. Conf. Detection of Intrusions and Malware and Vulnerability Assessment,* July 2005.

[29] G. Wassermann and Z. Su, "An Analysis Framework for Security in Web Applications," *Proc. FSE Workshop Specification and Verification of Component-Based Systems,* pp. 70-78, Oct. 2004.

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

[31] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," *Proc. 15th Usenix Security Symp.,* Aug. 2006.

**William G.J. Halfond** received the BS degree in computer science from the University of Virginia, Charlottesville, and the MS degree in computer science from the Georgia Institute of Technology, Atlanta, in May 2004. He is currently working toward the PhD degree in computer science in the College of Computing at the Georgia Institute of Technology. His research work is in software engineering and security, with emphasis on techniques that can be applied to improve the security and survivability of large-scale computer systems.

**Alessandro Orso** received the MS degree in electrical engineering and the PhD degree in computer science from Politecnico di Milano, Italy, in 1995 and 1999, respectively. He was a visiting researcher in the Department of Electrical Engineering and Computer Science at the University of Illinois at Chicago in 1999. Since March 2000, he has been with the College of Computing at the Georgia Institute of Technology, first as a research faculty member and then as an assistant professor. His area of research is software engineering, with emphasis on software testing and program analysis. His research interests include the development of techniques and tools for improving software reliability, security, and trustworthiness and the validation of such techniques on real systems. He is a member of the IEEE Computer Society.

**Panagiotis Manolios** received the BS and MA degrees in computer science from Brooklyn College, New York, in 1991 and 1992, respectively, and the PhD degree in computer science from the University of Texas at Austin in 2001. He joined the College of Computing at the Georgia Institute of Technology in 2001. He became an adjunct assistant professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology in 2003. He is currently an associate professor at Northeastern University. His main research interest is mechanized formal verification and validation. His other areas of interest include programming languages, distributed computing, logic, software engineering, algorithms, computer architecture, aerospace, and pedagogy. He is a member of the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.