SQLPrevent: Effective Dynamic Detection and Prevention of SQL Injection Attacks Without Access to the Application Source Code

San-Tsai Sun[*], Konstantin Beznosov[†]

Laboratory for Education and Research in Secure Systems Engineering
lersse.ece.ubc.ca
University of British Columbia
Vancouver, Canada

Technical report LERSSE-TR-2008-01[‡]

Last Modification Date: 2008/02/22

Revision: #14

[*]santsais@ece.ubc.ca
[†]beznosov@ece.ubc.ca
[‡]This and other LERSSE publications can be found at lersse-dl.ece.ubc.ca

Abstract

This paper presents an effective approach for detecting and preventing known as well as novel SQL injection attacks. Unlike existing approaches, ours (1) is resistant to evasion techniques, such as hexadecimal encoding or inline comment, (2) does not require analysis or modification of the application source code, (3) does not need training traces, (4) does not require modification of the runtime environment, such as PHP interpreter or JVM, and (5) is independent of the back-end database used.

Our approach is based on two simple observations, that (1) in malicious HTTP requests, parameter values are used not only as *literals* in the corresponding SQL statements but also as other SQL constructs, such as delimiters, identifiers or operators; and (2) a malformed parameter value in an HTTP request comprises more than one SQL *token*. We use J2EE to implement a tool we have named SQLPrevent that dynamically detects SQL injection attacks using the above heuristics, and blocks the corresponding SQL statements from being submitted to the back-end database. Using the AMNESIA testbed, we evaluate SQLPrevent over 15,000 unique HTTP requests with five web applications. In our experiments, SQLPrevent produced no false positives or false negatives, and imposed at most 4% (0.3% on average) performance overhead with respect to average 500 millisecond response time in the testbed applications.

Contents

1 Introduction **1**

2 Background **2**
2.1 How SQL Injection Attacks Work 3
2.2 Existing Countermeasures 4

3 Related Work **5**

4 Approach **7**
4.1 Abstraction of Web Applications and HTTP Requests 7
4.2 Alteration of the SQL Statement's Intended Syntactical Structure by SQLIAs 9
4.3 False Positive Reduction 10
4.4 Detection of Attacks 11

5 Evaluation **12**
5.1 Implementation 12
5.2 Experimental Evaluation 14

6 Discussion **17**

7 Conclusion **19**

References **20**

1 Introduction

An SQL injection attack *(SQLIA)* is a type of attack on web applications that exploits the fact that input provided by web clients is directly included in the dynamically generated SQL statements. SQLIA is one of the foremost threats to web applications [HVO06]. According to the WASP Foundation, injection flaws, particularly SQL injection, were the second most serious web application vulnerability type in 2007 [Pro07]. Since they are easy to find and exploit, SQL injection vulnerabilities are frequently employed by attackers .

The threats posed by SQLIAs go beyond simple data manipulation. Attackers commonly extract sensitive data (e.g., credit card information) or modify the content of the databases from the compromised web sites. Through SQLIAs, an attacker may also bypass authentication, escalate privileges, execute a denial-of-service attack, or execute remote commands to transfer and install malicious software. As a consequence of SQLIAs, parts of or whole organizational IT infrastructures can be compromised. An effective and easy to employ method of preventing SQLIAs is crucial for the protection of today's organizations.

Traditional SQLIA countermeasures are not effective [Anl02a, Anl02b, Cer03] and most web applications deployed today are still vulnerable to SQLIAs. The reasons are manyfold:

- SQLIAs are performed through HTTP traffic, sometimes over SSL, thereby making network firewalls ineffective.
- Defensive coding practices require training of developers and modification of the legacy applications to assure the correctness of validation routines and completeness of the coverage for all sources of input.
- Sound security practices—such as the enforcement of the principle of least privilege or attack surface reduction—can mitigate the risks to a certain degree, but they are prone to human error, and it is hard to guarantee their effectiveness and completeness.
- Signature-based web application firewalls—which act as proxy servers filtering inputs before they reach web applications—and other intrusion detection methods may not be able to detect SQLIAs that employ evasion techniques [Anl02a, Anl02b, Cer03].

Detection or prevention of SQLIAs is a topic of active research in industry and academia. Security Gateway [SS02] and commercial web application firewalls [AQT07, Inc07], implemented as proxy servers to prevent malicious input reaching vulnerable web applications, can be deployed without modifying the existing web applications. However, these tools suffer from both false positives and false negatives. An accuracy of 100% was claimed in recently published techniques that use static and/or dynamic analysis [HO05, BWS05, SW06, BBMV07], dynamic taint analysis [NTGG+05, PB05], or machine learning methods [VMV05]. However, the requirements for analysis and/or instrumentation of the application source code [HO05, BWS05, SW06, BBMV07], runtime environment modification [NTGG+05, PB05], or acquisition of training data [VMV05] limit the adoption

of these techniques in real-world settings.

In this paper, we propose a method for detecting and preventing SQLIAs at runtime. HTTP requests and the corresponding SQL statements are intercepted and analyzed. Detected SQLIAs are prevented by rejecting the HTTP requests that carry them. Our approach is capable of detecting novel obfuscated SQLIAs, and can be integrated with existing web applications without modifications to the applications. Our method does not require acquisition of training data, or modification of the runtime environment, such as PHP interpreter or JVM. Our approach is based on the following two simple observations, which we made after collecting and analyzing SQLIAs from white papers, technical reports, web advisories, web sites, and mailing lists:

1. In a benign HTTP request, parameter values are used only as *literals* in the corresponding SQL statements.
2. Each of those parameter values in an HTTP request that carries an SQLIA contains more than one SQL *token*.

We used J2EE to implement a tool we have named SQLPrevent that dynamically detects SQLIA using the above heuristics and blocks the corresponding SQL statements from being submitted to the back-end database. We evaluated SQLPrevent using the AMNESIA [HO05] testbed, which has been used for evaluating several other research systems (e.g., [SW06, BBMV07, KKH+07]). The testbed consists of five web applications and traces that contain about 3,000 malicious and 600 benign HTTP requests for each application. In addition to the attack inputs that come with the testbed, we created another set of about 3,000 obfuscated attack inputs per application, by applying the evasion techniques of hexadecimal encoding, dropping white spaces, and inserting inline comments to those from the testbed. In our experiments, SQLPrevent produced no false positives or false negatives. It imposed little performance overhead (maximum 4%, average 0.3%) with respect to 500 milliseconds response time in the testbed applications. The experimental results suggest that our technique is effective and efficient. Furthermore, SQLPrevent can be easily integrated with existing web applications, with only a few changes in the web server configuration settings.

The rest of the paper is organized as follows. Section 2 explains how SQL injection attacks and typical countermeasures work. Section 3 reviews existing work and compares it with the proposed approach. Section 4 describes our approach for detecting and preventing SQL injection attacks. Section 5 presents the evaluation methodology and results. Section 6 discusses the implications of the results and the strengths and limitations of our approach. Section 7 summarizes the paper and outlines future work.

2 Background

In this section, we explain how SQLIAs work and what countermeasures are currently available. Readers familiar with SQLIAs can proceed directly to the next section.

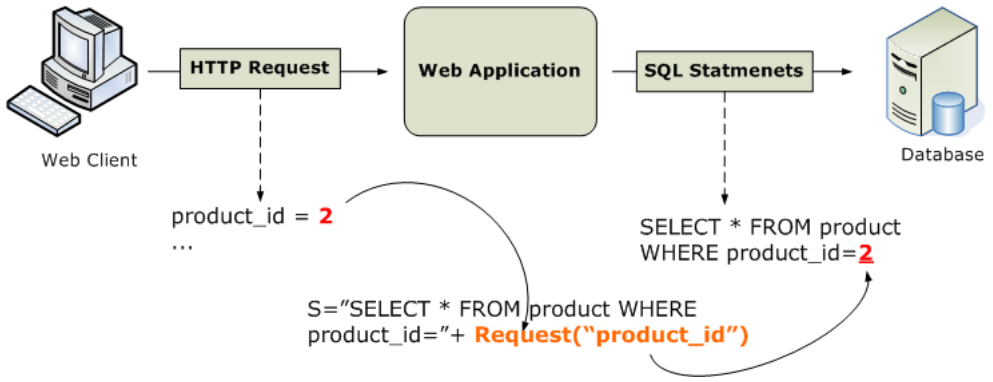

Figure 1: How SQL injection attacks work.

2.1 How SQL Injection Attacks Work

For the purpose of discussing SQLIAs, a web application can be thought of as a black box that accepts HTTP requests as inputs and generates SQL statements as outputs. Figure 1 depicts a simple set up with a web client sending an HTTP request to an application, which constructs and submits SQL statements to the back-end database. As shown in the sample code of the figure, web applications commonly use parameter values from HTTP requests to form SQL statements. SQLIAs may occur when data in an HTTP request is directly used to construct SQL statements without sufficient validation or sanitization. For instance, consider the following code (in Java) for constructing an SQL statement in which the value of the HTTP request parameter `product_id` is used directly in the statement:

```
S="SELECT * FROM product WHERE id=" + request.getParameter("product_id")
```

When the above line of code is executed in the web application, the value of the HTTP request parameter `product_id` is used in the SQL statement without any validation. By taking advantage of this vulnerability, an attacker can launch various types of attacks by simply posting HTTP requests that contain arbitrary SQL statements. Below is an example of a malicious HTTP request that modifies `price` information in a `product` table by appending the attack string "`update product set price=price/2`" to the legitimate input `product_id=2`, as shown in the following fragment:

```
POST /prodcut.jsp HTTP/1.1
product_id=2; update product set price=price/2
```

In the case of the above attack, the SQL statement constructed by the programming logic would be the following:

```
SELECT * FROM product WHERE id=2; update product set price=price/2
```

The above SQL statement would reduce the `price` of every `product` by one-half. The threats posed by SQLIAs can go beyond simple data manipulation. Consider the privilege escalation attack listed below :

```
POST /prodcut.jsp HTTP/1.1
product_id=2; exec master..xp_cmdshell 'net user hacker 1234 /add
```

If the injected code is executed by the database server, this attack would add a new user account named "hacker" with a password "1234" to the underlying Windows operating system. More malicious attacks, such as file upload and remote command execution, are also possible with similar attack techniques [Cer03].

To confuse signature-based detection systems, attackers may also apply evasion techniques that obfuscate attack strings. Below is an obfuscated version of the above privilege-escalation attack.

```
POST /prodcut.jsp HTTP/1.1
product_id=2; /* */declare/* */@x/* */as/* */varchar(4000)/* */set/* */
@x=convert(varchar(4000),0x6578656320206D61737465722E2E
78705F636D647368656C6C20276E657420757365722068616361636B6572
202F6164642027)/**/exec/* */(@x)
```

The above obfuscation utilizes hexadecimal encoding, dropping white space, and inline comment techniques. For a sample of evasion techniques employed by SQLIAs, see [MS05].

2.2 Existing Countermeasures

Because SQLIAs are carried out through HTTP traffic, even sometimes when it is protected by SSL, most traditional intrusion prevention mechanisms, such as firewalls or network-packet based intrusion detection systems (IDSs), are not capable of detecting SQLIAs. Three types of countermeasures are commonly used to prevent SQLIAs: web application firewalls, defensive coding practice, and service lock-down.

Web application firewalls such as WebKnight [AQT07] and ModSecurity [Inc07] are easy to deploy and operate. They are commonly implemented as proxy servers that intercept and filter HTTP requests before requests are processed by web applications. However, such tools are prone to both false positives and negatives. Due to the limitation of signature databases or policy rules, they may not effectively detect unseen patterns or obfuscated attacks that employ evasion techniques. Also, since those tools rely solely on analyzing HTTP requests and do not know the syntactic structures of the generated SQL statements, false positives might occur if signatures or filter policy rules are too restrictive.

Defensive coding practices are the primary basic prevention mechanism against SQLIAs [HL03]. Since the root cause of an SQLIA is insufficient user input validation, the most intuitive way to prevent SQLIAs is to sanitize inputs by validating input types, limiting input length, and checking user input for single quotes, SQL keywords, special characters, and other known malicious patterns. Using a parameterized query API provided by development platforms is another compelling solution for mitigating SQLIAs directly in code. Bound and typed parameters are used in parameterized queries, such as `PrepareStatement` in

Java and `SQLParameter` in .NET. Parameterized queries syntactically separate the intended structure of SQL statements and data literals. Instead of composing SQL statements by simply concatenating strings, each parameter in an SQL statement is declared using a placeholder, and the corresponding literal value for each placeholder is then provided separately.

Service lock-down is employed to limit the damage resulting from SQLIAs. System administrators can create least-privileged database accounts to be used by web applications, configure different accounts for different tasks and reduce unused system procedures. However, similar to defensive coding practices, these countermeasures are prone to human error, and it is difficult to assure their correctness and/or completeness.

Having discussed the state of the practice, in the next section we provide an overview of the state of the art.

3 Related Work

Research work related to SQLIA detection or prevention can be broadly categorized based on the type of data analyzed or modified by the proposed techniques: (1) runtime HTTP requests, (2) design-time web application source code and (3) runtime dynamically generated SQL statements. To detect SQLIAs, some approaches use only one type of data while others use two. For example, our approach analyzes HTTP requests and SQL statements. Below we discuss related work using these categorizations, and briefly summarize the advantages and limitations of each. For a more detailed discussion, we refer the reader to a classification of SQLIA prevention techniques in [HVO06].

Runtime filtering of HTTP requests: Security Gateway [SS02] is a filtering proxy that allows only those HTTP requests that are compliant with the input validation rules to reach the protected web applications. Like commercial web application firewalls, Security Gateway is easy to deploy and operate, without any modifications to the application source code. However, this approach requires developers to provide correct validation rules, which are specific to their application. Similarly to the defensive programming practices, this process requires intimate knowledge of the web application in question; as a result, it is prone to false positives and false negatives. Also, any modification of an existing web application or deployment of a new one requires modification to the input validation rules, leading to an increase in the administrative and change management overheads. Our approach does not need developer involvement and requires deployment of interception modules only when a new instance of a web application is deployed.

Web application source code analysis and hardening: WebSSARI [HYH+04] and approaches proposed by Livshits et al. [LL05] and Xie et al. [XA06] use information-flow-based source code analysis techniques to detect SQLIA vulnerabilities in web applications. Once detected,

these vulnerabilities can be fixed by the developers. These approaches to vulnerability detection employ static analysis of applications. They have the advantages of no runtime overhead and the ability to detect errors before deployment; however, they need access to the application source code, and the analysis has to be repeated each time an application is modified. Such access is sometimes unrealistic, and repeated analysis increases the overhead of change management. Our approach does not require access to the source code and is oblivious to application modification.

Runtime analysis of SQL statements for anomalies: Valuer et al. [VMV05] propose an SQLIA detection technique based on machine learning methods. Their anomaly-based system learns profiles of the normal database access performed by web-based applications using a number of different models. These models allow for the detection of unknown attacks with limited overhead. After learning "normal" profiles in a training phase, the system uses deviation from these profiles to detect potential attacks. Valuer et al. have shown that their system is effective in detecting SQLIAs. However, the fundamental limitation of this and other approaches based on machine learning techniques is that their effectiveness depends on the quality of training data used. Training data acquisition is an expensive process and its quality may not be guaranteed. Our approach does not rely on the ability of the application developers or owners to acquire a qualified "clean" data set—which has all possible versions of legitimate SQL statements and yet has no SQLIAs.

Static analysis paired with runtime analysis of SQL statements: AMNESIA [HO05], SQLGuard [BWS05], SQLCheck [SW06], and CANDID [BBMV07] identify the intended structures of SQL statements by analyzing the source code of web applications at development time and checking at runtime whether dynamically generated SQL statements conform to those structures. SQLrand [BK04] modifies SQL statements in the source code by appending a randomized integer to every SQL keyword during design-time; an intermediate proxy intercepts SQL statements at runtime and removes the inserted integers before submitting the statements to the back-end database. Therefore, any normal SQL code injected by attackers will be interpreted as an invalid expression. These approaches are very effective, claiming 100% accuracy (i.e., no false positives and no false negatives). Like the other approaches discussed above ([HYH+04, LL05, XA06]), the SQLIA prevention solutions in this class need access to the application source code for the purpose of analysis and modification, which is their main limitation.

Runtime analysis of HTTP requests and SQL statements: Approaches employing dynamic taint analysis have been proposed by Nyguyen-Tuong et al. [NTGG+05] and Pietraszek et al. [PB05]. Taint information refers to data that come from un-sanitized or un-validated sources, such as HTTP requests. Both approaches modify the PHP interpreter to mark tainted data as it enters the application and flows around. Before any database access function, e.g., `mysql_query()`, is dispatched, the corresponding SQL statement string is checked by the modified PHP interpreter. If tainted data has been used to cre-

ate SQL keywords and/or operators in the query, the call is rejected. Similar to our technique, these approaches use HTTP requests and SQL statements, do not require access to the application source code, do not need training traces, and are resistant to evasion techniques. Their limitations are that they (1) require modifications to the PHP runtime environment, which may not be viable for other runtime environments such as Java or ASP.NET, and (2) need all database access functions to be identified in advance. Our approach has neither limitation.

Sania [KKH+07], an SQLIA vulnerability testing tool, identifies injectable parameters by comparing the parse trees and HTTP responses for a benign HTTP request and the corresponding auto-generated attack. The main drawback of this approach is the high rate of false positives (about 30%) and the need for application developers to be involved in the SQLIA vulnerability testing.

4 Approach

Our approach is based on (1) abstracting a web application as a function that takes HTTP requests as inputs and generates SQL statements as outputs, (2) abstracting an HTTP request as a set of name-value pairs, (3) making particular observations about the alteration of the intended syntactical structure of the dynamically generated SQL statements by SQLIAs, and (4) observations about how false positives can be reduced.

4.1 Abstraction of Web Applications and HTTP Requests

For the purpose of discussing SQLIAs, we abstract a web application as a function that takes HTTP requests as inputs and generates SQL statements as outputs. We exclude from our observation communications made by web applications to other data sources such as XML documents, LDAP servers or arbitrary files IOs. Since only HTTP requests, and not responses, can carry an SQLIA payload, we also exclude HTTP responses from further discussion.

A web client requests services by making an HTTP request to a web server. An HTTP request message consists of the following three parts, as illustrated in Figure 2:

Request line with optional query strings, such as:
`POST /bookstore/book.jsp?ACTION=UPDATE&book_id=123 HTTP/1.1`
It requests the file `book.jsp` from `bookstore` directory with query strings `ACTION=UPDATE&book_id=123`

Headers, such as `Accept-Language:en-us` and `User-Agent:Mozilla/4.0`. The character "`:`" is used to separate the name and value of a header. Note that the cookie header is commonly abstracted as a separate object due to its unique purpose. Cookies are opaque strings of text sent by a server to a web browser that are stored locally on the client and then sent back unchanged by the browser each time it accesses that server. HTTP cookies are commonly used

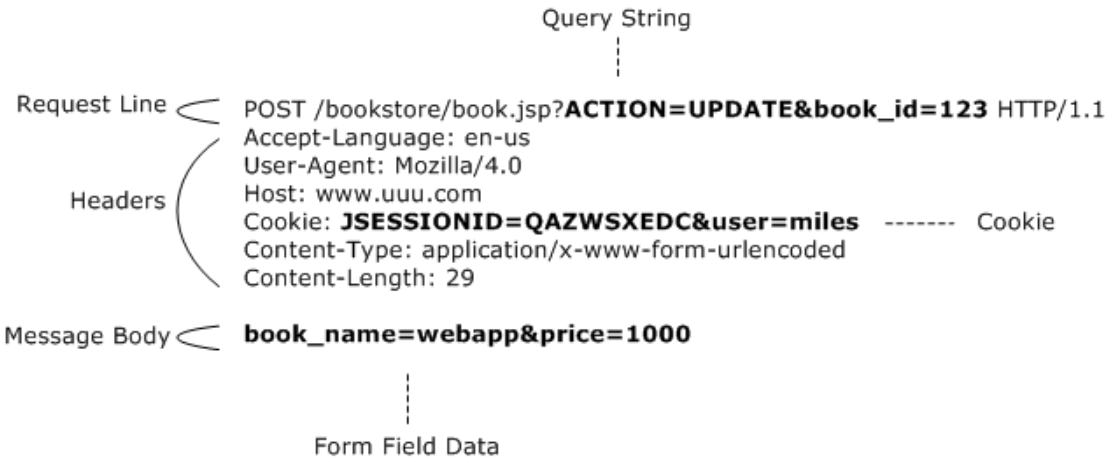

Figure 2: Structure of an HTTP request and sources of name-value pairs.

Table 1: Abstraction of HTTP request from the example in Figure 2.

Source	**Name (n)**	**Value (v)**
Query String	ACTION	UPDATE
Query String	book_id	123
Cookie	JSESSION_ID	QAZWSXEDC
Cookie	user	miles
Header	Accept-Language	en-us
Header	User-Agent	Mozilla/4.0
Form Data	book_name	webapp
Form Data	price	1000

for authenticating, session tracking, and maintaining specific information about users.

Message body. This is an optional part of an HTTP request. When the POST method is used, the message body consists of user input data in an HTML form, such as `book_name=webapp&price=1000`

We abstract an HTTP request in the context of SQLIAs as a *set of name-value pairs in which the name part serves as a identifier for a given input parameter.* There are four possible sources of input parameters in an HTTP request: (1) query string, (2) cookie collection, (3) header collection, and (4) form field data. For example, the HTTP request from Figure 2 can be abstracted as shown in Table 1. Thus, we can represent an HTTP request as an element of a powerset of parameters, 2^P, where each element of P is a 2-tuple (n, v) of name and value.

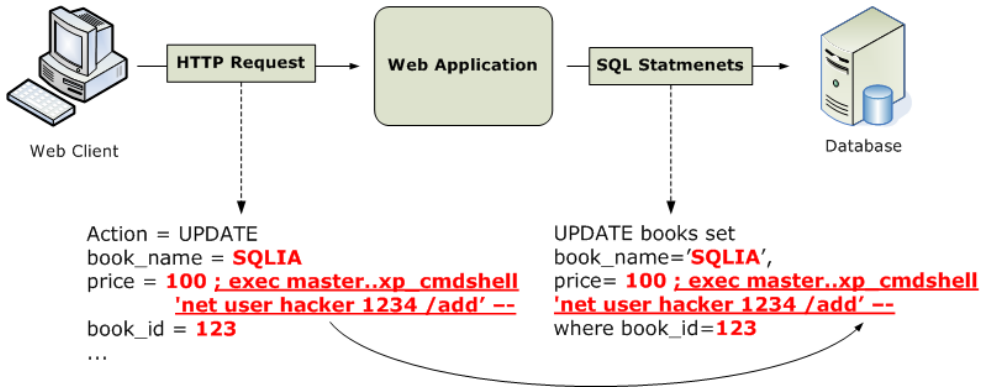

Figure 3: An attacker tries to inject an additional SQL statement into original query.

4.2 Alteration of the SQL Statement's Intended Syntactical Structure by SQLIAs

Our first key observation is that *in a benign HTTP request, parameter values are used only as literals in the corresponding SQL statements.* An SQL literal is a notation for representing a fixed value within an SQL statement. For example, in the given SQL statement `UPDATE books set book_name='webapp', price='1000' WHERE book_id=123"`, the literals are "`webapp`" for `book_name` column , "`1000`" for `price` column and "`123`" for `book_id` column. Our detection heuristic identifies those cases where parameter values of an HTTP request show up in the corresponding SQL statements as something other than literals. We now explain why this observation can be considered as a general rule for dynamic detection of SQLIAs.

Web application developers typically use string manipulation functions to dynamically compose SQL statements by concatenating pre-defined constant strings with parameter values from HTTP requests. Given the sample HTTP request in Figure 2, the following Java code constructs an SQL statement by embedding parameter values from query string (`book_id`) and form field data (`book_name` and `price`):

```
statement= "UPDATE books set " +
           "book_name='"+request.getParameter("book_name")+ "',"+
           "price="+request.getParameter("price")+ " "
           "WHERE book_id="+ request.getParameter("book_id");
```

This scenario is a typical case of coding database access logic in web applications. The intended syntactical structure of the SQL statement in the above example can be expressed as follows: `"UPDATE books set book_name=?, price=?  WHERE book_id=?"`, where question marks are used as placeholders for the parameter values. When the placeholders are instantiated with parameter values, those values should only be used as *literals* in order to maintain the original syntactical structure of the SQL statement. Otherwise, adversaries can launch attacks by injecting extra single quotes, SQL keywords, operators, or delimiters into the SQL statements to alter the syntactical structure of SQL statements.

Here is a simple example. As shown in Figure 3, an attacker tries to inject an additional SQL statement into the original query by using query delimiter (";") and comment characters ("– –") that mark the beginning of a comment. As a result, instead of just updating `book_name` and `price` information for books whose `book_id` equals `123`, an attack in Figure 3 causes the application to update `book_name` to "`webapp`" and `price` to `1,000` for every entry in the `books` table, and also adds a new user account named "`hacker`" with a password "`1234`" to the underlying MS Windows operating system.

4.3 False Positive Reduction

Based on our first observation, false positives may occur when a parameter value appears in the corresponding dynamic SQL statement, but is not actually used by the programming logic in the process of composing the final SQL statement. Consider the example in Figure 4. The parameter named `ACTION` has a value of "`UPDATE`", which appears in the dynamic SQL statement; however, the "`UPDATE`" is taken from a pre-defined constant string instead of the HTTP request. If only examining whether it is a literal according to our first observation, the example above would be an occurrence of a false positive—a benign request being categorized as a malicious attack, since "`UPDATE`" is not a literal in the final SQL statement.

The second key observation employed in our approach is that *the HTTP request parameter value that carries an SQLIA string requires at least two SQL tokens for the attack to work*—one for the original placeholder value and another for the attack. An SQL token is a categorized block of text, such as keyword (i.e., SELECT, UPDATE and FROM), string literal, identifer (e.g., book_id and book_name columns) or operator (i.e., +,- and =). Since one token is insufficient for an attack, SQLIAs comprise more than one token. Among over 500 distinct SQLIA strings we investigated, the shortest attack string we found was a numeric literal value followed by a shutdown command, such as `2 SHUTDOWN` where `2` and `SHUTDOWN` are two distinct tokens separated by a white space. The resulting attack query would look like the following: `SELECT book_name from books WHERE book_id=2 SHUTDOWN`.

The fact that a malicious parameter value requires at least two SQL tokens to launch an attack is an important property for eliminating false positives when performing SQLIA detection. Since web applications do not automatically provide information about the source of tokens in the dynamic SQL statements, it is not clear whether a specific token is from pre-defined strings or from an HTTP request. By using the number of tokens in a parameter as a threshold value, false positives could be significantly reduced. In fact, when we evaluated SQLPrevent using `two` as the threshold value, of 3,824 benign HTTP requests from the AMNESIA [HO05] testbed, none caused a false positive. Note that false positives may be still possible even if the threshold for the number of tokens in a parameter value is `two`. We delay this discussion until Section 6 when we address the limitations of our approach.

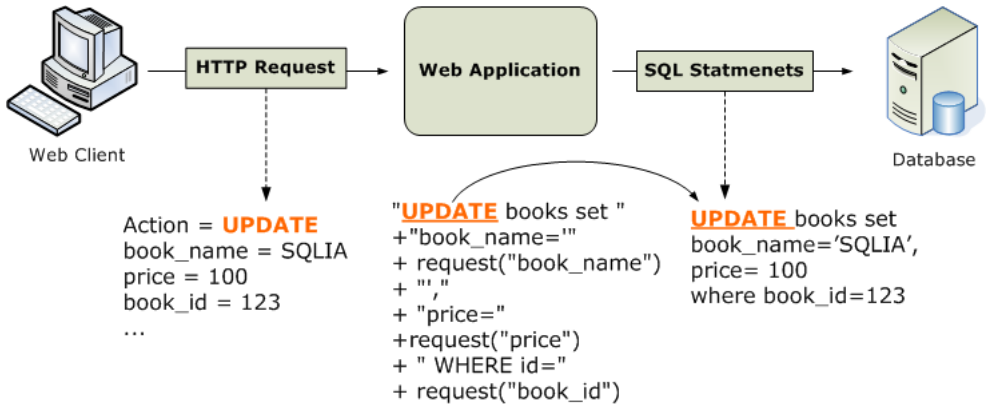

Figure 4: An example of a false positive: keyword UPDATE is from constant string instead of HTTP request

4.4 Detection of Attacks

Using the above observations and the abstractions of a web application and an HTTP request, we developed two heuristics for detecting SQLIAs. To summarize our heuristics, SQLIAs occur when (1) parameter values within an HTTP request are used to construct SQL statements in such a way that the parameter values modify the intended syntactical structure of the dynamic SQL statements, and (2) a malicious parameter value contains at least two SQL tokens.

Based on the above heuristics, we developed an algorithm to detect whether an intercepted HTTP request is an SQLIA. Algorithm 1 takes an HTTP request r and an SQL statement string s as inputs and returns `true` if r is malicious, otherwise returns `false`. The algorithm determines whether r is an SQLIA attack by checking if there is a parameter value in r that is a substring of the intercepted SQL statement but is not in the set of literal values of s, and contains at least two SQL tokens.

input : A set of parameter strings r in an intercepted HTTP request
input : An intercepted SQL statement string s
output: A boolean value indicate whether r is malicious or not

$\triangle \leftarrow$ set of literal tokens in s
for every p *in* r **do**
  $t \leftarrow$ number of tokens in p
  if p *is substring of* s ***and*** $p \notin \triangle$ ***and*** $t > 1$ **then**
    return true
  end
end
return false

Algorithm 1: IsHTTPRequestMalicious

To analyze the computational complexity of Algorithm 1, let N be the number of

parameters in an HTTP request, M the length in characters of the longest parameter, and L the length of the SQL statement in characters. The detection algorithm loops through N parameters in the HTTP request in question. For each parameter, it counts the number of tokens within the parameter and performs a substring search against the SQL statement in question. Finding the number of tokens in a parameter (line 3) requires reading through each character in it, thus the complexity for this operation is $O(M)$. For substring search in line 4, the complexity is $O(M + L)$ according to [Sun90]. We assume the operator $\notin$ used in line 4 takes constant time if the literal tokens are first put into a hash table. Thus, the overall computational complexity of Algorithm 1 is $O(N(M + L))$.

5 Evaluation

To evaluate our approach, we developed a tool named SQLPrevent that implements Algorithm 1, and evaluated it using the testbed suite from project AMNESIA [HO05]. We chose this testbed because it allowed us to have a common point of reference with other approaches that have used it for evaluation [SW06, BBMV07, KKH+07].

5.1 Implementation

SQLPrevent is implemented in J2EE platform and consists of an HTTP request intercepter, thread-local storage, SQL interceptor, SQLIA detector, and SQL lexer modules. As illustrated in Figure 5, the original data flow (HTTP request → web application → JDBC driver → database) is modified when SQLPrevent is deployed into a web server. First, the references to the program objects representing incoming HTTP requests are saved into the current thread-local storage. Second, the SQL statements composed by web applications are intercepted by the SQL interceptor and passed to the SQLIA detector module. The detection module then retrieves the corresponding HTTP request from thread-local storage and examines the request to determine whether it contains an SQLIA. If so, the SQL interceptor prevents the malformed SQL statement from being submitted to the database. All main modules of SQLPrevent are shown in Figure 5, and are explained below.

HTTP Request interceptor is implemented as a *servlet filter*—a component type introduced in Java Servlet specification version 2.3 [Cow01]. This module intercepts HTTP requests and stores an internal reference to the object representing the intercepted HTTP request in the corresponding thread-local storage. The stored reference is retrieved later by the SQLIA detector module when it processes the intercepted SQL statements.

Thread-local storage is static or global memory local to a thread—each thread gets a unique instance of thread-local static or global variables. Given that web servers are commonly implemented as multi-threaded processes that handle multiple concurrent HTTP requests at the same time, the SQLIA detector module needs a way to find the corresponding HTTP request for each intercepted SQL statement. Since both request handling and query generation are

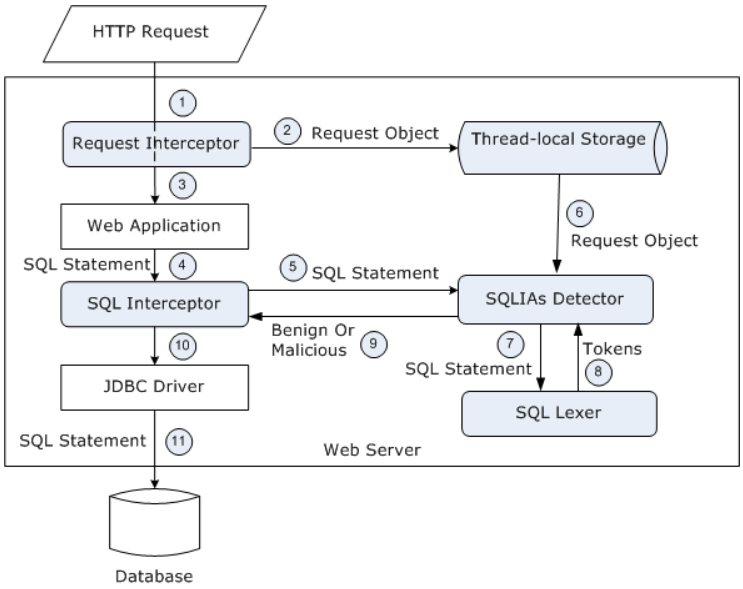

Figure 5: Main elements of SQLPrevent architecture are shown in light blue/grey. The data flow is depicted with sequence numbers and arrow labels.

processed in the same thread, the thread-local storage provides an adequate mechanism for a one-to-one mapping between an HTTP request and the corresponding SQL statement.

SQL interceptor extends P6Spy [MGAQ03]. This open-source module intercepts and logs SQL statements issued by web-application programming logic before they reach the JDBC driver. We have extended P6Spy to invoke the SQLIA detector when SQL statements are intercepted.

SQLIA detector takes an intercepted SQL statement as input, retrieves the corresponding HTTP request object from the thread-local storage, passes the intercepted SQL statement to the SQL lexer for tokenization, and then performs detection according to Algorithm 1. If an SQLIA is identified, the detector indicates this fact to the SQL interceptor, which throws a necessary security exception to the web application, instead of letting the SQL statement through.

SQL lexer is implemented as a lexical analyzer. This module converts a sequence of characters into a sequence of tokens. The SQL lexer module is used to perform lexical analysis of intercepted SQL statements. Given an SQL statement, the SQL lexer generates a set of tokens with the corresponding token types. For example, by giving the following SQL statement as an input: "`UPDATE books SET book_name='SQLIA', price=100 WHERE book_id=123`", the SQL lexer will generate the following set of tokens and the corresponding token types:

No.	Token	Token Type
1.	UPDATE	[IDENTIFIER]
2.	books	[IDENTIFIER]
3.	SET	[IDENTIFIER]
4.	book_name	[IDENTIFIER]
5.	=	[OPERATOR - EQUALS]
6.	'SQLIA'	[LITERAL - STRING]
7.	,	[COMMA]
8.	price	[IDENTIFIER]
9.	=	[OPERATOR - EQUALS]
10.	100	[LITERAL - INTEGER]
11.	WHERE	[IDENTIFIER]
12.	book_id	[IDENTIFIER]
13.	=	[OPERATOR - EQUALS]
14.	123	[LITERAL - INTEGER]

The SQL lexer is used by the SQLIA detector module to find a set of literal types in the intercepted SQL statement, such as `LITERAL - STRING` in line `6` and `LITERAL - INTEGER` in line `10` and line `14`.

The source code of SQLPrevent consists of 2,009 lines of actual code, of which the lexical analyzer constitutes just over 80% of the code base.

5.2 Experimental Evaluation

To evaluate SQLPrevent, we used the testbed suite from AMNESIA [HO05] and set up the experimental environment as illustrated in Figure 6. The testbed suite consists of an automatic testing script in Perl and five web applications (Bookstore, Employee Directory, Classifieds, Events, and Portal). Each web application came with an ATTACK list of about 3,000 malformed inputs and a LEGIT list of over 600 legitimate inputs. In addition to the original ATTACK lists, we produced another set of obfuscated attack lists by obscuring original attack inputs using hexadecimal encoding, dropping white space, and inline comments evasion techniques to validate the ability of SQLPrevent to detect obfuscated SQLIAs. To test whether SQL lexer module is capable of performing lexical analysis in a database-independent way, we configured Microsoft SQL Server and MySQL as back-end databases. SQLPrevent was tested with each of the five applications and each of the two databases, resulting in ten test runs.

To make sure the performance measurements were not skewed by fast hardware, we used low-end equipment. The web applications and databases were installed on a machine with a 1.8 GHz Intel Pentium 4 processor and 512 MB RAM, running Windows XP SP2. The automatic test script was executed on a host with a 350 MHz Pentium II processor and 256 MB of memory, running Windows 2003 SP2. These two machines were connected over a local area network with 100 Mbps Ethernet adapters to minimize the network delays. Round-trip latency, while pinging the server from the client machine, was less than 1 millisecond on average.

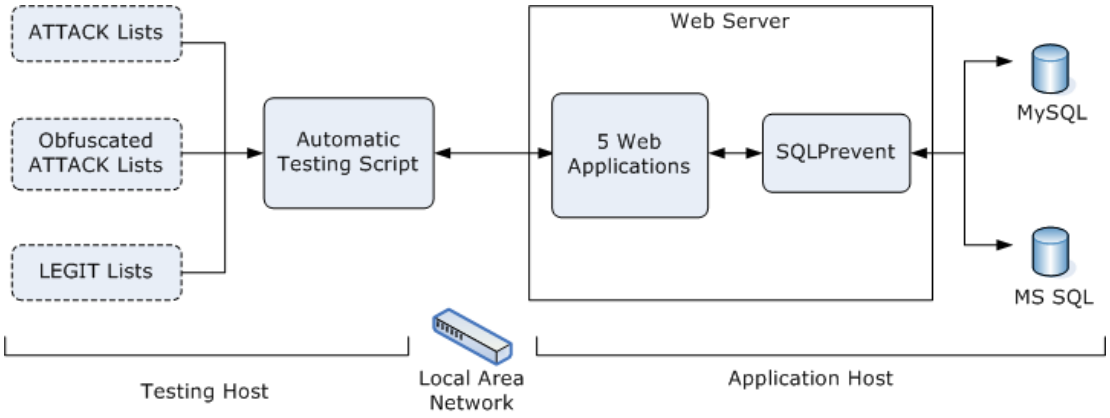

Figure 6: Evaluation Environment Setup.

SQLIA detector threw an exception (`java.sql.SQLException`) each time it detected an attack. The testbed web applications embedded the exception message into the HTTP response before replying to the web client. By examining the SQLIA exception message in the HTTP response, the automatic testing script was able to determine whether a test input was recognized as malicious or not.

In our experiments, we subjected SQLPrevent to a total of 3,824 benign and 15,876 malicious HTTP requests. We also obfuscated the requests carrying SQLIAs and tested SQLPrevent against them, which resulted in doubling the number of malicious requests. We then repeated the experiments using an alternative back-end database. In total, we tested SQLPrevent with over 70,000 HTTP requests. None of these requests resulted in SQLPrevent producing a false positive or false negative.

To measure the performance characteristics of SQLPrevent, we used nanosecond API in J2SE 1.5 and provided two sets of evaluation data. The first set was used for measuring *detection overhead*, which is the time delay imposed by SQLPrevent for each benign HTTP request. To calculate *detection overhead*, we measured the round-trip response time with and without SQLPrevent for each benign HTTP request, as shown in Figure 7, and applied the following formula: $\textit{Detection Overhead} = (t_b - t)/t$, where t_b and t are round-trip (between A to C in Figure 7) response times with and without SQLPrevent respectively.

The second set of data was for measuring *prevention overhead*, which is the overhead imposed by SQLPrevent when a malicious SQL statement is blocked. *Prevention overhead* shows how fast SQLPrevent can detect and prevent an SQLIA. If either overhead is too high, the system could be vulnerable to denial-of-service attacks that aim for resource over-consumption. To ensure that SQLPrevent would not impose high overhead when blocking SQLIAs, we conducted another performance test and used the following formula to calculate *prevention overhead*: $\textit{Prevention Overhead} = (t_r + t_s)/t_m$, where t_r and t_s are the time delays for request interceptor and SQL interceptor, respectively, and t_m is round-trip (from A to B) response time when a malicious SQL statement is blocked.

Table 2 shows, for each web application and the corresponding database, the max-

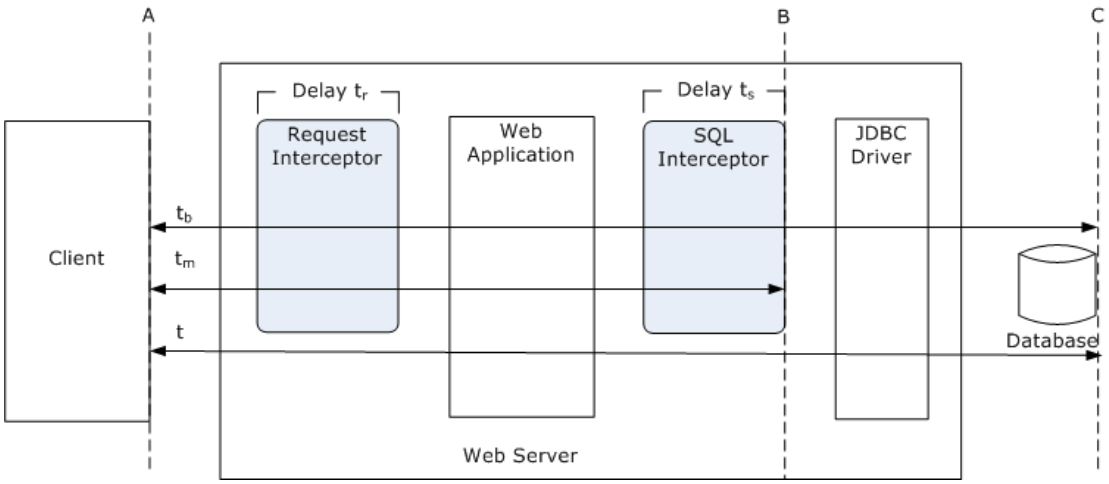

Figure 7: Detection and prevention performance evaluation. t_b and t_m are round-trip response time with SQLPrevent deployed, measured using benign and malicious requests, respectively.

imum, minimum, and average *detection overhead* and *prevention overhead*. SQLPrevent imposed a maximum 4% (average 0.3%) performance overhead with respect to an average 500 milliseconds response time for all five applications and both databases. The overhead for blocking detected SQLIAs is lower than in the case of benign requests likely because in the former case the SQL statements are not executed by the back-end database.

To test SQLPrevent performance overhead under a high volume of simultaneous accesses, we used JMeter [Fou07], a web application benchmarking tool from Apache Software Foundation. For each application, we chose one servlet and configured 100 concurrent threads with five loops for each thread. Each thread simulated one web client. We then measured the average response time with and without SQLPrevent and applied the *detection overhead* formula to calculate the overhead. During stress testing, SQLPrevent imposed a maximum 4.2% (average 2.6%) performance overhead with respect to an average 6,700 milliseconds response time for all five applications and both databases.

Due to the differences in physical settings, we cannot compare SQLPrevent performance directly with other approaches that also use the AMNESIA testbed. Therefore, we list the performance data of the latter here for reference purposes only. AMNESIA [HO05] simply stated that "*We found that the overhead imposed by our technique is negligible and, in fact, barely measurable, ranging from 10 to 40 milliseconds*" without detailed information regarding the physical settings and how overhead was measured. The SQLCheck [SW06] evaluation environment was set up on a machine running Linux kernel 2.4.27, with a 2 GHz Pentium M processor and 1 GB of memory. The timing results were presented in a table, and the average overhead for each application ranged from 2.478ms to 3.368ms. Nevertheless, the table did not show maximum overhead information and the paper did not state how the per-

Table 2: SQLPrevent overheads for cases with benign ("detection") and malicious ("prevention") HTTP requests.

DB	Subject	Detection Overhead (%)			Prevention Overhead (%)		
		Max	Min	Ave	Max	Min	Ave
MS SQL	Bookstore	3.632	0.028	0.617	2.113	0.074	0.216
	Employee	2.894	0.029	0.171	2.151	0.022	0.227
	Classifieds	3.343	0.014	0.228	1.987	0.057	0.212
	Events	4.038	0.028	0.257	2.442	0.064	0.392
	Portal	3.685	0.025	0.545	1.703	0.047	0.145
MySQL	Bookstore	2.561	0.019	0.355	2.457	0.069	0.244
	Employee	3.754	0.031	0.412	2.461	0.068	0.246
	Classifieds	2.671	0.036	0.023	1.757	0.062	0.249
	Events	3.943	0.024	0.051	2.051	0.016	0.237
	Portal	3.896	0.033	0.038	1.616	0.045	0.201
		4.038	***0.014***	***0.271***	***2.461***	***0.016***	***0.237***
		$\approx$ ***4.0***		$\approx$ ***0.3***	$\approx$ ***2.5***		$\approx$ ***0.2***

formance overhead was measured. CANDID [BBMV07] was evaluated by installing web applications on a Linux machine with a 2GHz Pentium processor and 2GB of RAM. The machine ran in the same Ethernet network as the client. Using JMeter, one servlet was chosen from each application, and a detailed test suite was prepared for each application. For each test, the researchers performed 1,000 sample runs and measured the average numbers for each run with and without CANDID, respectively. Results were shown in a figure, and ranged from 3.2% to 40.0%.

6 Discussion

In our evaluations, SQLPrevent produced no false positives or false negatives, imposed low runtime overhead on the testbed applications, and was portable among two different databases. Some existing approaches [HO05, BWS05, SW06, BBMV07, NTGG+05, PB05] also have either low performance overhead or high accuracy. However, compared with SQLPrevent, they suffer from other limitations, such as the need to analyze or even modify the application source code [HO05, BWS05, SW06, BBMV07] or to modify the runtime environment [NTGG+05, PB05].

In spite of the compelling evaluation results, our approach could in theory have false positives or false negatives, since web applications do not automatically provide information about the source of tokens in the dynamic SQL statements. Based on our detection algorithm, a false positive would occur when a parameter value in an HTTP request (1) appears as a substring of the intercepted SQL statement and (2) is not in the literal token set of the intercepted SQL statement and (3) comprises more than two tokens, and (4) is not used by programming logic to form the SQL statement. For example, in Figure 4, if the parameter named `ACTION` had a value

of "`UPDATE books`", this would be an instance of a false positive for our detection algorithm. However, as shown by the evaluation, our detection algorithm correctly identified all 3,824 benign requests we had in the testbed, by ruling out parameters that comprise only one token. The chances of false positives could be further reduced by simply configuring the threshold values (i.e., the number of tokens in the parameter value) for that particular URL in the SQLIA detector, at the cost of an additional configuration.

Theoretically, false negatives are also possible in our approach, since a web application could use the value of an HTTP request parameter in any way it wants when it constructs the SQL statement. For instance, consider a parameter value that consists of a list of comma-delimited product categories `categories=c1,c2`, and assume that the server-side programming logic constructs a separate SQL statement for each category id in the list, such as:

```
id_array = request.getParameter("categories").split(",");
S1="SELECT * FROM category WHERE cid='"+id_array[0]+"'";
S2="SELECT * FROM category WHERE cid='"+id_array[1]+"'";
```

A malicious parameter `"categories=c1,c2' shutdown --"` could successfully exploit this vulnerability, resulting in S_2 as `"SELECT * FROM category WHERE cid='c2' shutdown"`. This attack would not be detected by our detection algorithm, since the whole malformed parameter value (`"c1,c2' shutdown --"`) is not a substring of S_2.

To generalize the above example, false negatives can occur when a malformed parameter value in an HTTP request (1) is modified by web application programming logic before it is used to construct the final SQL statement or (2) is partially selected by programming logic to form the SQL statement. Since both conditions result in a malicious parameter not appearing as a substring of the intercepted SQL statement, the malformed parameter will be neglected by our detection algorithm. However, based on the experimental results and to the best of our knowledge, these are rare cases; the most common cause of SQLIAs is programming logic using malicious parameters directly to form SQL statements without any validation or modification. For those rare cases, an extension module that performs a customized parsing logic can be configured to be used by SQLPrevent before performing detection. For instance, the above false negative sample can be prevented by an extension that splits the value of "`categories=c1,c2`" into separate parameters such as "`categories_1=c1`" and "`categories_2=c2`" before the detection module commences detection. Thoroughly addressing the problems of false positives and false negatives will be a candidate subject of future research.

In addition to high detection accuracy and low performance overhead, the advantages of our technique are its ease of integration with existing web applications and databases, and its portability across different back-end databases. SQLPrevent can be easily integrated with existing web applications based on J2EE technology by simply (1) deploying SQLPrevent Java library into J2EE application servers, (2) configuring *HTTP request interceptor* filter entry in the `web.xml`, and (3) replacing

the class name of the real JDBC driver with the class name of *SQL interceptor*. Our approach requires web servers to have capabilities for performing HTTP request filtering and SQL statement interception. For SQLPrevent, we implemented the HTTP request interceptor module as a *filter* and SQL interceptor module as a *JDBC proxy*. The *filter* was introduced in Java Servlet specification version 2.3 [Cow01] and *JDBC* has been part of the Java Standard Edition since the release of SDK 1.1. To the best of our knowledge, most J2EE application servers support both API interfaces these days. We are currently working on the port of SQLPrevent to ASP.NET and PHP for the purpose of evaluating the feasibility of our approach for these mainstream web environments.

Our approach also appears to be compatible with different back-end databases. Most database-system vendors develop proprietary SQL dialects (such as Microsoft T-SQL [Cor07a], Oracle PL-SQL [Cor07c] or MySQL [Cor07b]) in addition to supporting standard ANSI SQL. To protect different types of back-end databases against SQLIAs, an SQLIA detection mechanism that utilizes an SQL parsing technique (such as SQLGuard [BWS05], SQLCheck [SW06], CANDID [BBMV07] and Sania [KKH+07]) must provide SQL parsers that support each type of SQL dialect. SQL parsing, or syntactic analysis, is the process of analyzing a sequence of tokens to determine its grammatical structure with respect to a given SQL grammar. Even though most existing SQL grammars are substantially different from each other, they all share similar lexical rules for tokenizing an SQL statement. Our approach uses an SQL lexical analyzer instead of an SQL grammar parser to analyze intercepted SQL statements, which makes any implementation based on our approach easier to port to other back-end databases. For instance, SQLPrevent is used with MySQL without any modification to the SQL lexer, which was originally designed for Microsoft SQL Server.

7 Conclusion

SQL injection vulnerabilities are ubiquitous and dangerous, yet most web applications deployed today are still vulnerable to SQLIAs. Although recent research on SQLIA detection and prevention has successfully addressed the shortcomings of existing SQLIA countermeasures, the effort needed from web developers—such as application source code analysis/modification, acquisition of the training traces, or modification of the runtime environment—has limited adoption of these countermeasures in real world settings. In this paper, we have presented a new approach to runtime SQLIA detection and prevention, as well as a tool (SQLPrevent) that implements our approach. Our evaluation of SQLPrevent indicates that it is effective, efficient, portable among back-end databases, easy to deploy without the involvement of web developers, and does not require access to the application source code.

For future work, we plan to conduct additional research to thoroughly address the problems of false positives and/or false negatives. We also plan to finish porting our approach to other web-application development platforms, such as ASP.NET and PHP, in order to evaluate the feasibility of our approach for other mainstream web platforms. To obtain more realistic data on the practical possibility of false positives

and false negatives, we plan to evaluate SQLPrevent on other real world web applications, and test it with SQLIA penetration testing tools such as Absinthe [NX07] and SQLNinja [ice07]. We will also make SQLPrevent an open source project.

Acknowledgments

We thank William Halfond and Alex Orso for providing AMNESIA [HO05] testbed applications and sample attacks for use in our evaluation, and Craig Wilson for improving the readability of the paper. Members of the Laboratory for Education and Research in Secure Systems Engineering (LERSSE) supplied valuable feedback on the earlier drafts of this paper. Special thanks go to Kirstie Hawkey and Kasia Muldner for their detailed suggestions on improving this paper.

References

[Anl02a] Chris Anley. Advanced SQL injection in SQL server application. *Technical report, NGSSoftware Insight Security Research (NISR)*, 2002.

[Anl02b] Chris Anley. (more) Advanced SQL injection in SQL server application. *Technical report, NGSSoftware Insight Security Research (NISR)*, 2002.

[AQT07] AQTRONIX. WebKnight. http://www.aqtronix.com/?PageID=99, 2007.

[BBMV07] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. In *ACM Conference on Computer and Communications Security (CCS)*, pages 12–24, Alexandria, Virginia, USA, October 2007.

[BK04] Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Second International Conference on Applied Cryptography and Network Security (ACNS)*, pages 292–302, June 2004.

[BWS05] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. SQLGuard: Using parse tree validation to prevent SQL injection attacks. In *International Workshop on Software Engineering and Middleware*, pages 106–113, Lisbon, Portugal, September 2005.

[Cer03] Cesar Cerrudo. Manipulating Microsoft SQL server using SQL injection. *Technical report, Application Security Inc.*, 2003.

[Cor07a] Microsoft Corp. Transact-SQL reference. http://msdn2.microsoft.com/en-us/library/ms189826.aspx, 2007.

[Cor07b] MySQL AB Corp. MySQL 6.0 reference manual. http://dev.mysql.com/doc/refman/6.0/en/index.html, 2007.

[Cor07c] Oracle Corp. Oracle database PL/SQL. http://www.oracle.com/technology/tech/pl_sql/index.html, 2007.

[Cow01] Danny Coward. JSR-000053: Java Servlet specification, version 2.3. Specification v.2.3 Final Release, Java Community Program, September 2001.

[Fou07] Apache Software Foundation. Apache JMeter. http://jakarta.apache.org/jmeter/, 2007.

[HL03] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, Washington, 2nd edition, 2003.

[HO05] William G.J. Halfond and Alessandro Orso. AMNESIA: Analysis and monitoring for neutralizing SQL injection attacks. In *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183, Long Beach, California, USA, 2005.

[HVO06] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL injection attacks and countermeasures. In *IEEE International Symposium on Secure Software Engineering*, 2006.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D˙ T˙Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *13th international conference on World Wide Web*, pages 40–52, 2004.

[ice07] icesurfer. SQLNinja. http://sqlninja.sourceforge.net/, 2007.

[Inc07] Breach Security Inc. ModSecurity. http://www.modsecurity.org/, 2007.

[KKH+07] Yuji Kosuga, Kenji Kono, Miyuki Hanaoka, Miho Hishiyama, and Yu Takahama. Sania: Syntactic and semantic analysis for automated testing against SQL injection. In *23rd Annual Computer Security Applications Conference (ACSAC)*, December 2007.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *14th USENIX Security Symposium*, pages 271–286, August 2005.

[MGAQ03] Andy Martin, Jeff Goke, Alan Arvesen, and Frank Quatro. P6Spy open source software. http://www.p6spy.com/, 2003.

[MS05] Ofer Maor and Amichai Shulman. SQL injection signatures evasion. *White Paper of Imperva Inc.*, 2005.

[NTGG+05] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, pages 296–307, Makuhari-Messe, Chiba, Japan, May 30 - June 1 2005.

[NX07] Nummish and Xeron. Absinthe. http://www.0x90.org/releases/absinthe/, 2007.

[PB05] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Eighth International Symposium on Recent Advances in Intrusion Detection*, pages 124–145, 2005.

[Pro07] Open Web Application Security Project. OWASP top 10 threats in web application 2007. http://www.owasp.org/index.php/Top_10_2007, 2007.

[SS02] David Scott and Richard Sharp. Abstracting application-level web security. In *11th International Conference on the World Wide Web*, pages 396–407, Honolulu, Hawaii, USA, May 2002.

[Sun90] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33:132–142, 1990.

[SW06] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Symposium on Principles of Programming Languages*, pages 372–382, Charleston, South Carolina, USA, January 2006.

[VMV05] Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A learning-based approach to the detection of SQL attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005)*, pages 123–140, 2005.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium*, pages 179–192, August 2006.