Retrofitting Existing Web Applications with Effective Dynamic Protection Against SQL Injection Attacks

San-Tsai Sun and Konstantin Beznosov
Department of Electrical and Computer Engineering
University of British Columbia

Abstract

This paper presents an approach for retrofitting existing web applications with runtime protection against known as well as unseen SQL injection attacks (SQLIAs) without the involvement of application developers. The precision of the approach is also enhanced with a method for reducing the rate of false positives in the SQLIA detection logic, via runtime discovery of the developers' intention for individual SQL statements made by web applications. The proposed approach is implemented in the form of protection mechanisms for J2EE, ASP.NET, and ASP applications. Named SQLPrevent, these mechanisms intercept both HTTP requests and SQL statements, mark and track parameter values originating from HTTP requests, and perform SQLIA detection and prevention on the intercepted SQL statements. The AMNESIA testbed is extended to contain false-positive testing traces, and is used to evaluate SQLPrevent. In our experiments, SQLPrevent produced no false positives or false negatives, and imposed a maximum 3.6% performance overhead with 30 milliseconds response time for the tested applications.

Introduction

SQL injection attacks *(SQLIAs)* are one of the foremost threats to web applications (W. G. Halfond, Viegas, & Orso, 2006). According to the WASP Foundation, injection flaws, particularly SQL injection, were the second most serious type of web application vulnerability in 2008 (OWASP, 2008). The threats posed by SQLIAs go beyond simple data manipulation. 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 entire organizational IT infrastructures can be compromised. As a case in point, SQLIAs were apparently employed by Ehud Tenenbaum, who has been arrested on charges

We thank William Halfond and Alex Orso for providing AMNESIA (W. G. Halfond & Orso, 2005) 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.

of stealing $1.5M from Canadian and at least $10M from US banks (Zetter, 2009). An effective and easy to employ method for protecting numerous existing web applications from SQLIAs is crucial for the security of today's organizations.

State-of-the-practice SQLIA countermeasures are far from effective (Anley, 2002) and many web applications deployed today are still vulnerable to SQLIAs (OWASP, 2008). 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 network-level intrusion detection methods may not be able to detect SQLIAs that employ evasion techniques (Maor & Shulman, 2005).

Detection or prevention of SQLIAs is a topic of active research in industry and academia. An accuracy of 100% is claimed by recently published techniques that use static and/or dynamic analysis (W. G. Halfond & Orso, 2005; Buehrer, Weide, & Sivilotti, 2005; Su & Wassermann, 2006; Bandhakavi, Bisht, Madhusudan, & Venkatakrishnan, 2007), dynamic taint analysis (Nguyen-Tuong, Guarnieri, Greene, Shirley, & Evans, 2005; Pietraszek & Berghe, 2005), or machine learning methods (Valeur, Mutz, & Vigna, 2005). However, the requirements for analysis and/or instrumentation of the application source code (W. G. Halfond & Orso, 2005; Buehrer et al., 2005; Su & Wassermann, 2006; Bandhakavi et al., 2007), runtime environment modification (Nguyen-Tuong et al., 2005; Pietraszek & Berghe, 2005), or acquisition of training data (Valeur et al., 2005) limit the adoption of these techniques in some real-world settings. Moreover, a common deficiency of existing SQLIA approaches based on analyzing dynamic SQL statements is in defining SQLIAs too restrictively, which leads to a higher than necessary percentage of false positives (FPs). False positives could have significant negative impact on the utility of detection and protection mechanisms, because investigating them takes time and resources (Julisch & Darcier, 2002; Werlinger, Hawkey, Muldner, Jaferian, & Beznosov, 2008). Even worse, if the rate of FPs is high, security practitioners might become conditioned to ignore them.

In this paper, we propose an approach for retrofitting existing web applications with runtime protection against known as well as unseen SQL injection attacks (SQLIAs) without the involvement of application developers. Our work is mainly driven by the practical requirement of web-application owners that a protection mechanism should be similar to a software-based security appliance that can be "dropped" into an application server at any time, with low administration and operating costs. This "drop-and-use" property is vital to the protection of web applications where source code, qualified developers, or security development processes might not be available or practical.

To detect SQLIAs, our approach combines two heuristics. The first heuristic (labeled as "token type conformity") triggers an alarm if the parameter content of the corresponding HTTP request is used in non-literal tokens (e.g., identifiers or operators) of the SQL statement. While efficient, this heuristic leaves room for false positives when the application developer (intentionally or accidentally) includes tainted SQL keywords or operators in a dynamic SQL statement. This case would trigger an SQLIA alarm, even though the query does not result in an SQLIA. For instance, as a common case of result-set sorting, a developer could *intentionally* include a predefined parameter

value in an HTTP request to form an "`ORDER BY`" clause in an SQL statement. As we explain later in the paper, the existing approaches and the detection logic based solely on the first heuristic would trigger an SQLIA alarm because the keywords "`ORDER`" and "`BY`" are tainted, even though the intercepted SQL statement is indeed benign. In this case, the user is supplying input intended by the programmer; she is not *injecting* SQL.

When a potential SQLIA is detected by the first heuristic, our approach employs the second heuristic (labeled as "conformity to intention") to eliminate the above type of false positives. We put forward a new view of an SQLIA: an attack occurs when the SQL statement produced by the application at runtime does not conform to the syntactical structure intended by the application developer. Intention conformity enables runtime discovery of the developers' intention for individual SQL statements made by web applications. Defined more precisely later in the paper, such a view of an SQLIA requires "reverse engineering" of the developer's intention. Our approach not only "discovers" the intention but does so at runtime, which is critical for those applications that are provided without source code. To discover the intended syntactical structures, our approach performs dynamic taintness tracking at runtime and encodes the intended syntactical structure of a dynamic query in the form of SQL grammar, which we term *intention grammar*. Our detection algorithm triggers an alarm if the intercepted SQL statement does not conform to the corresponding intention grammar.

To evaluate our approach, we developed SQLPrevent. It is a software-based security appliance that (1) intercepts HTTP requests and SQL statements at runtime, (2) marks parameter values in HTTP requests as tainted, (3) tracks taint propagation during string manipulations, and (4) performs analysis of the intercepted SQL statements based on our heuristics. To evaluate SQLPrevent, we employed the AMNESIA (W. G. Halfond & Orso, 2005) testbed, which has been used for evaluating several other research systems. We extended the AMNESIA testbed to contain requests with new false positives, and added another set of obfuscated attack inputs per application. In our experiments, SQLPrevent produced no false positives or false negatives, and imposed little performance overhead (maximum 3.6%, standard deviation 1.4%), with 30 milliseconds response time for the tested applications.

The rest of the paper is organized as follows. In the next section, we explain how SQL injection attacks and typical countermeasures work. Then we review existing work and compare it with the proposed approach. We then describe our approach in detail for detecting and preventing SQL injection attacks. Next, we discuss the implementation of SQLPrevent in J2EE, ASP.NET, and ASP, followed by a description of the evaluation methodology and results. Finally, we discuss the implications of the results and the strengths and limitations of our approach before summarizing the paper and outlining future work.

Background

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

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, as illustrated in Figure 1.

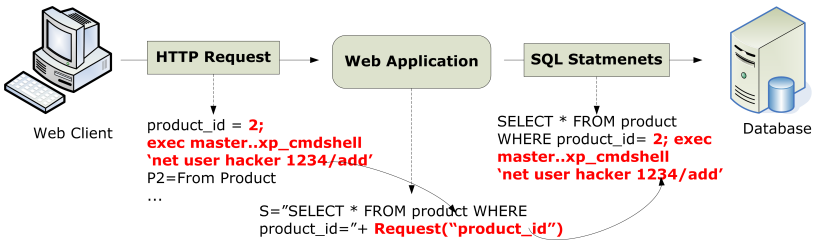

Figure 1. How SQL injection attacks work.

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, when `S="SELECT * FROM product WHERE id=" + request.getParameter("product_id")` 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 posting HTTP requests that contain arbitrary SQL statements. Below is an example of a malicious HTTP request:

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

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; _
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 (Anley, 2002).

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),0x6578656320206D6173
7465722E2E78705F636D647368656C6C20276E6574207573657220686
636B6572202F6164642027)/**/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 (Maor & Shulman, 2005).

False Positives

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. In these cases, programmers can freely incorporate user inputs to form dynamic SQL statements. Without taking developers' SQL-grammatical intentions into account, false positives are possible in all existing dynamic SQLIA approaches. We illustrate this false-positive problem through a running example.

Example 1 *Assume there is an HTML dropdown list named "order_by", which consists of three entries—"*`without order`*", "*`by id`*", "*`by name`*". Each entry and its corresponding value is shown in the following HTML code:*

```
<select name='order_by'>
   <option value=''>without order</option>
   <option value='ORDER BY id'>by id</option>
   <option value='ORDER BY name'>by name</option>
</select>
```

Assume a programmer intentionally *uses the value of the parameter "*`order_by`*" to form an SQL query, as illustrated in the following Java code fragment:*

S="SELECT c1 FROM t1" + request.getParameter("order_by");

Based on a user's selection at runtime (assume the second entry is selected), the SQL statement constructed by the above programming logic would be "`SELECT c1 FROM t1 ORDER BY id`*", where underlined labels indicate the data originated from an HTTP request.*

Obviously, the above Java code fragment is vulnerable. An attacker can launch an arbitrary attack by simply appending an attack string to the legitimate input "`order_by=ORDER BY id`". However, during normal operations, the dynamically constructed SQL statements are indeed benign and harmless.

Existing Countermeasures

Because SQLIAs are carried out through HTTP traffic, sometimes protected by SSL, most traditional intrusion-prevention mechanisms, such as firewalls or signature-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 practices, and service lock-down.

Web application firewalls such as WebKnight (AQTRONIX, 2007), ModSecurity (Breach Security Inc., 2007) and Security Gateway (Scott & Sharp, 2002) 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, 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, false positives might occur if signatures or filter policy rules are too restrictive.

Defensive coding practices are the most intuitive ways to prevent SQLIAs, by validating input types, limiting input length, or checking user input for single quotes, SQL keywords, special characters, and other known malicious patterns. Using a parameterized query API (e.g., `PrepareStatement` in Java and `SQLParameter` in .NET) is another compelling solution for mitigating SQLIAs directly in code, as parameterized queries syntactically separate the intended structure of SQL statements and data literals.

Service lock-downs are procedures 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 un-used 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.

Related Work

Existing research 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. Below, we discuss related work using this categorization, briefly summarize the advantages and limitations of existing approaches, and demonstrate why false positives are possible in some approaches. For a more detailed discussion, we refer the reader to a classification of SQLIA prevention techniques in (W. G. Halfond et al., 2006) and our technical report (Sun & Beznosov, 2009).

Web application source code analysis and hardening: WebSSARI (Huang et al., 2004), and approaches proposed by Livshits and Lam (2005), Jovanovic, Kruegel, and Kirda (2006), and Xie and Aiken (2006) use information-flow-based static analysis techniques to detect SQLIA vulnerabilities in web applications. Once detected, these vulnerabilities can be fixed by the developers. 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.

Runtime analysis of SQL statements for anomalies: Valeur et al. (2005) propose an SQLIA detection technique based on machine learning methods. 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 cannot be guaranteed. Non-perfect training data causes such techniques to produce false positives and false negatives.

Static analysis with runtime protection: SQLrand (Boyd & Keromytis, 2004) 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. For our running Example 1 of false positive, the intercepted SQL statement in SQLrand would read as "`SELECT`^{key} `c1 FROM`^{key} `t1 ORDER BY id`", where "`key`" represents the random key. The intercepted SQL statement would cause a false positive, since the keywords "`ORDER`" and "`BY`" are not appended with the random key.

SQLGuard (Buehrer et al., 2005) provides programmers with a Java library to manually bracket the placeholders of user input in SQL statements. During runtime, SQLGuard compares two parse trees of the dynamically created SQL statement with and without input values respectively. In the case of Example 1, SQLGuard will compare parse trees of (1) "SELECT c1 FROM t1 *key*ORDER BY id*key*", and (2) "SELECT c1 FROM t1 *keykey*", where the first query contains input value and the second does not. SQLGuard would trigger an alarm for this query since neither augmented query is a valid SQL statement.

AMNESIA (W. G. Halfond & Orso, 2005) builds legitimate SQL statement models using static analysis based on information flow. At runtime, SQL statements that do not conform to the corresponding pre-built model are rejected and treated as SQLIAs. Since the automaton of the model "SELECT $\rightarrow$ c1 $\rightarrow$ FROM $\rightarrow$ t1 $\rightarrow$ β" would not accept the example dynamic SQL (corresponding β must be string or numeric constant), the SQL query from Example 1 would be an instance of false positive in AMNESIA.

WASP (W. G. J. Halfond, Orso, & Manolios, 2006) prevents SQLIAs by checking whether all SQL keywords and operators in an SQL statement are marked as trusted. To track trusted sources, WASP uses Java byte-code instrumentation techniques to mark all hard-coded and implicitly created strings in the source code, and strings from external sources (e.g., file, trusted network connection, database) as trusted. In the case of Example 1, WASP would view the intercepted SQL statement as " SELECT c1 FROM t1 ORDER BY id", where underlined labels indicate the data are trusted. Since the keywords "ORDER" and "BY" are not marked as trusted, the query would be rejected as an instance of false positive.

SQLCheck (Su & Wassermann, 2006) detects SQLIAs by observing the syntactic structure of generated SQL queries at runtime, and checking whether this syntactic structure conforms to an augmented grammar. The main limitation of SQLCheck is that it requires each parameter value to be augmented with the meta-characters in order to determine the source of substrings in the constructed SQL statement. This approach requires manual intervention of the developer to identify and annotate untrusted sources of input, which introduces incompleteness problems and may lead to false negatives. In addition, wrapping meta-characters around each parameter value might cause unexpected side-effects. For instance, if the programming logic in a web application performs string comparison using the augmented parameter value, the result would be different than in the case of no meta-characters, which would cause unexpected results in business logic (e.g., math operations of two user inputs). In addition, the generated SQL statement for Example 1 would read as "SELECT c1 FROM t1 $\triangleleft$ ORDER BY id $\triangleright$", where $\triangleleft$ and $\triangleright$ are special meta-characters added by SQLCheck. This query would be treated as an injection attack if the augmented grammar does not state user inputs are permitted in "ORDER" and "BY" keywords.

CANDID (Bandhakavi et al., 2007) transforms a Java web application by adding a benign candidate variable v_c for each string variable v. When v is initialized from the user-input, v_c is initialized with a benign candidate value that is the same length as v. If v is initialized by the program, v_c is also initialized with the same value. CANDID then compares the real and candidate parse trees at runtime. Using Example 1, the real and the corresponding candidate SQL statement would be "SELECT c1 FROM t1 ORDER BY id", and "SELECT c1 FROM t1 aaaaaaaaaaa", respectively. The intercepted SQL statement would be treated as an attack, since the parse trees derived from the two queries differ.

Runtime analysis of HTTP requests and SQL statements: Approaches employing dynamic taint analysis have been proposed by Nguyen-Tuong et al. (2005) and Pietraszek and Berghe

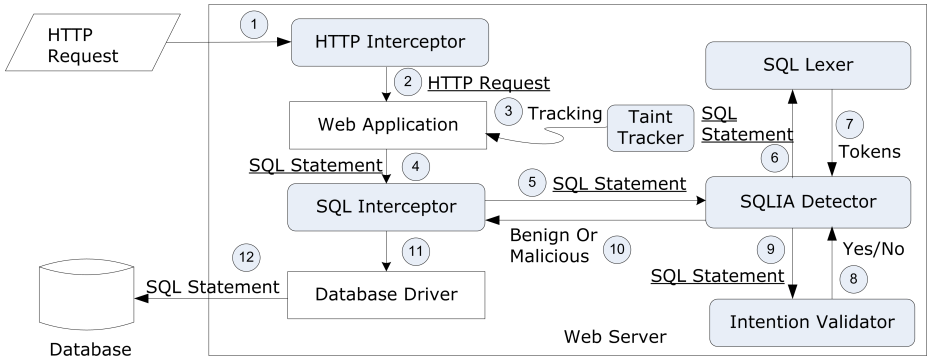

Figure 2. Main elements of SQLPrevent architecture are shown in light grey. The data flow is depicted with sequence numbers and arrow labels. Underlined labels indicate that the data are accompanied by the tainted meta-data. Depending on whether an SQL statement is benign or potentially malicious, data may flow to the Intention Validator conditionally.

(2005). 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 they enter the application and flow around. If tainted data have been used to create SQL keywords and/or operators in the query, the call is rejected. For the running example, the intercepted SQL statement would be viewed as "`SELECT c1 FROM t1 ORDER BY id`", where underlined labels indicate the data are tainted. Since the keywords "`ORDER`" and "`BY`" are marked as tainted, the query would be rejected—which is an instance of false positive. Sekar (2009) proposed a black-box taint-inference technique that infers tainted data in the intercepted SQL statements, and then employs syntax and taint-aware policies for detecting unintended use of tainted data. His technique achieves taint-tracking without intrusive instrumentation on target applications or modification to the runtime environment. However, false positives and false negatives are possible due to sub-optimal accuracy of the taint-inference algorithm and taint-awareness policies.

Approach

Our approach enables retrofitting existing web applications with run-time protection against known as well as unseen SQLIAs. The core of the approach is a software-based security appliance, SQLPrevent, which can be "plugged" into a web server without any modifications to the hosted web applications. As illustrated in Figure 2, SQLPrevent consists of `HTTP Interceptor`, `Taint Tracker`, `SQL Interceptor`, `SQL Lexer`, `Intention Validator`, and `SQLIA Detector` modules. When SQLPrevent is deployed in a web server, the original data flow (HTTP request → web application → database driver → database) is altered. First, the reference to the program object representing an incoming HTTP request is intercepted by `HTTP Interceptor`, and data in the request are marked as tainted. Second, propagation of tainted data is tracked by `Taint Tracker`. Finally, the SQL statements issued by web applications are intercepted by the `SQL Interceptor` and passed to the `SQLIA Detector`. The `SQLIA Detector` module performs detection based on the two heuristics (*token type conformity* and *conformity to intention*) to detect an attack. Token type conformity determines whether an HTTP request is benign or potentially malicious by checking whether tainted data are used only as string or numeric literals in the intercepted SQL statement. `SQL Lexer` is used by

`SQLIA Detector` module to tokenize SQL statements. Normally, most dynamically constructed SQL statements are benign. When a potential SQLIA is detected (i.e., any non-literal token contains tainted characters), `SQLIA Detector` passes a tainted SQL statement to `Intention Validator` to confirm whether tainted non-literal tokens have been intentionally constructed by developers. If the intercepted SQL statement does not conform to the intended syntactical structure, `SQLIA Detector`, depending on the configuration, either triggers an alarm or prevents the malformed SQL statement from being submitted to the database. Note that any HTTP request that violates toke type conformity will be flagged as a potential vulnerability. Whereas the SQLPrevent architecture is based on a standard approach of implementing a security subsystem in the form of interceptors, our approach is distinguished by its detection logic. The following subsections describe each of the detection heuristics in detail.

Token Type Conformity

The core of the token type conformity heuristic is based on the observation that SQLIAs always cause a parameter value, or its portion, to be interpreted by the back-end database as something other than an SQL string or numeric literal, thus altering the intended syntactical structure of the dynamically generated SQL statement. In order to retain statements' intended syntactical structure, however, parameter values from HTTP requests should be used only as SQL string or numeric literals.

Tracking of Tainted Data. Tainted data refers to data that originates from an untrusted source, such as an HTTP request. An SQLIA occurs when tainted data are used to construct an SQL statement in a way that alters the intended syntactical structure of the SQL statement. To trace the source of each character in an SQL statement for web applications, we designed per-character taint propagation using a custom implementation of Java's string-related classes. Our design (1) contains an additional data structure—referred as *taint meta-data*—for tracking the taint status of each character in a string, and (2) implements public methods for setting/getting the taint meta-data. This meta-data is propagated during string manipulations, such as concatenation, extraction, or conversion.

Lexical Analysis of SQL Statements. SQLPrevent performs lexical analysis of SQL statements at run-time in order to identify non-literal tokens in the SQL statements. Lexical analysis is the process of generating a stream of tokens from the sequence of input characters comprising the SQL statement. The goal of lexical analysis in our approach is to generate two sets of tokens: LITERALS and NON-LITERALS. The LITERALS set contains string and number tokens, and the NON-LITERALS set has tokens of all other types. The exact types of tokens in the NON-LITERAL set are irrelevant for the purpose of our detection logic. This simplified design of the lexical analyzer makes our approach efficient and more portable among databases. For instance, during the experiments, our implementation of SQL lexer worked with MySQL without any modification, even though the lexer was originally designed for Microsoft SQL Server.

Detecting SQLIAs. Applying our heuristic that parameter values should only be used as string or numeric literals in the dynamic SQL statements, the mechanisms of taint tracking, and SQL lexical analysis, we developed an algorithm for SQLIA detection using token type conformity. Shown in Algorithm 1, the algorithm takes an SQL statement s and taint information about the characters in s as a implicit parameter. If tainted character(s) appears in any non-literal token (e.g.,

Algorithm 1: Token type conformity SQLIA detection algorithm

Input: An intercepted SQL statement string s
Output: A boolean value indicates whether s is malicious or not
$\triangle \leftarrow$ set of tokens in s;
for every token t in $\triangle$ **do**
  if typeOf(t) $\neq$ string or number literal **and** isTainted(t) **then**
    return true;
  end if
end for
return false;

identifier, delimiter, or operator) of s, the algorithm returns `true`, otherwise `false`. For each token of an intercepted SQL statement, if the type of token is not a literal (i.e., not a string or number), and the token is tainted, then the intercepted SQL statement is potentially malicious.

The "token type conformity" heuristic was originally inspired by Perl taint mode (Wall, 2007). When in taint mode, the Perl runtime explicitly marks data originating from outside of a program as tainted. Tainted data are prevented from being used in any security sensitive functions such as shell commands, or database queries. To "untaint" an untrusted input, the tainted data must be passed through a sanitizer function written in regular expressions. However, developers have to manually untaint user input data, and sanitizer functions might not catch all malicious inputs, especially when evasion techniques are employed. Nguyen-Tuong et al. (2005) and Pietraszek and Berghe (2005) modified PHP interpreter to support taint tracking. The main limitation of their approach is that they require modifications to the PHP runtime environment and database access functions, which may not be viable for other runtime environments such as Java, ASP.NET or ASP.

The effectiveness of our approach depends on the precision of taint tracking. However, the traces of taint meta-data might be lost due to certain limitations in the tainting implementation. For instance, in Java, string-related classes export character-based functions (e.g., toCharArray) for retrieving internal characters of a string. The taint tracking module is unable to propagate taint meta-data to primitive types unless a modified version of JVM is employed. Thus, the taint information would be lost if an application constructs a new instance of string based on the internal characters of another string. Nevertheless, based on the experimental results and to the best of our knowledge, retrieving internal buffer of a string to construct an SQL statement is a rare case, and it is common coding practice that a programmer should validate any binary data retrieved from an unsafe buffer (Howard & LeBlanc, 2003).

Conformity to Intention

To protect the integrity of SQL statements, our token type conformity heuristic, and some existing approaches, use pre-defined taint policies, implicitly or explicitly, to specify where in an SQL statement the untrusted data are allowed, and then check at runtime whether an intercepted SQL statement conforms to those policies. Based on the pre-defined taint policies, these approaches employ various mechanisms to track tainted data, and distinguish them in a dynamic query. However, while these approaches are effective, by using static taint policies and not taking developers' intentions into account, false positives are possible (as we demonstrated in Example 1).

Instead of using pre-defined taint policies, we take the issue of explicit information-flow

```
select_statement ::= ``SELECT'' select_list from_clause [where_clause]
                     [order_clause]
select_list      ::= ``*'' | id_list
id_list          ::= ID | ID ``,'' id_list
from_cause       ::= ``FROM'' id_list
where_clause     ::= ``WHERE'' cond { (``AND'' | ``OR'' ) cond }
cond             ::= value OPERATOR value
value            ::= ID | STRING | NUMBER
order_clause     ::= ``ORDER BY'' id_list
```

Figure 3. A simplified SQL SELECT statement grammar written in Backus-Naur Form (BNF).

one step further, and treat SQLIA as a problem of detecting whether a given SQL query conforms with the original intention of the application developer. Our second heuristic, which we labeled as "conformity to intention," allows discovery of the intended syntactical structure of a dynamic SQL statement at runtime, and performing validation on the SQL statement against the dynamically identified intention. To the best of our knowledge, there is no dynamic SQLIA detection and prevention technique that employs a concept similar to "conformity to intention".

Intention Statement. Web application developers typically specify the intended syntactical structure of an SQL statement using *placeholders* directly in code. For instance, the following Java code constructs a dynamic SQL statement by embedding parameter values from an HTTP request (each parameter might also pass through a sanitizer function):

Example 2 Typical Java code for constructing an SQL statement with the use of an HTTP request object:

```
statement= "SELECT book_name," +  request.getParameter("p1")
  + " FROM " + request.getParameter("p2")
  + " WHERE book_id='" + request.getParameter("p3") + "' "
  + request.getParameter("p4");
```

The intended syntactical structure of the SQL statement in the above example can be expressed as shown in code Fragment 1, where an underlined question mark is used to indicate a placeholder:

```
"SELECT book_name,? FROM ? WHERE book_id='?' ?"          (1)
```

We refer to such a parameterized SQL statement as an *intention statement*. Our approach relies on per-character taint tracking for deriving intention statements during runtime. When an SQL statement is intercepted, our taint tracker marks every character in a token as tainted when the token contains one or more tainted characters. Our approach constructs an intention statement by replacing each consecutive tainted substring in a dynamically constructed SQL statement with a special meta-character. Thus, when the SQL statement "`SELECT book_name,price FROM book WHERE book_id='SQLIA' ORDER BY price`" is intercepted, our approach substitutes each tainted substring with the placeholder meta-character (?) to form an intention statement, as shown in code Fragment 1. Note that even when a statement containing an SQLIA, such as "`SELECT book_name, price FROM book WHERE book_id='SQLIA'`

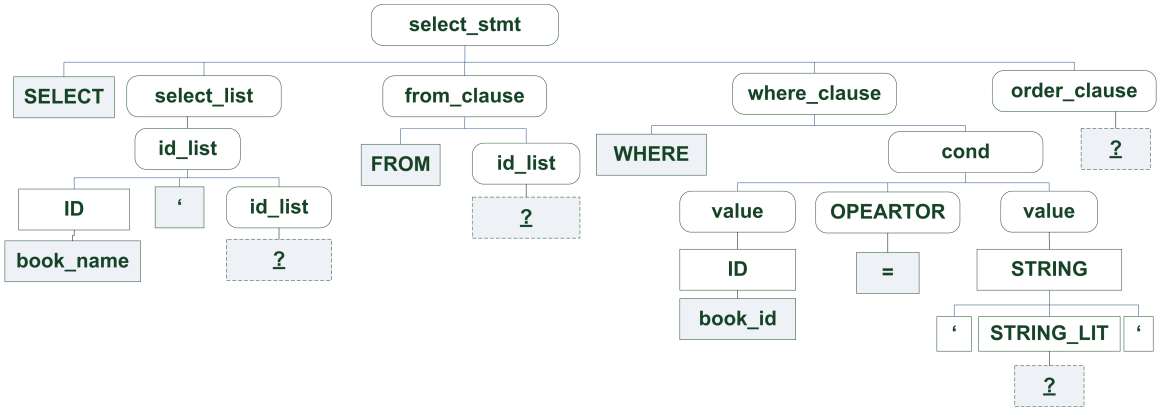

Figure 4. The intention tree of the intention statement from Fragment 1. Oval boxes represent *nonterminal symbols*, square boxes represent *terminal symbols*, and dash-lined boxes are placeholders. The grammar rules for each placeholder are (from left to right) two `id_lists`, a `STRING_LIT`, and an `order_clause`.

`ORDER BY price ; UPDATE users SET password=null`" is intercepted, the derived intention statement is the same as the one in code Fragment 1.

A placeholder in an intention statement represents an expanding point, where each expansion must conform to the corresponding grammatical rule intended by the developer. We denote a placeholder's corresponding grammar rule as an *intention rule*, which regulates the instantiation of a placeholder at runtime. Each intention rule maps to an existing nonterminal symbol (e.g., SELECT list) or terminal symbol (e.g., string literal or identifier) of a given SQL grammar. The collection of intention rules of an SQL statement serves as the intended syntactical structure, and can be discovered by using an SQL parse tree.

Intention Tree and Intention Grammar. An intention statement is a string without explicit structure. To identify the intention rules of an intention statement, we use an SQL parse tree. Our approach constructs a parse tree (referred to in this paper as an *intention tree*) from an intention statement to represent the explicit syntactical structure of an intention statement. Figure 4 illustrates an intention tree for the intention statement in Fragment 1, based on the simplified SQL SELECT statement sample grammar shown in Figure 3. The sample grammar consists of a set of production rules, each of the form $\alpha ::= \omega$, where α is a single *nonterminal* symbol, and ω is any sequence of *terminals* and/or *nonterminals*. In the example from Figure 3, the `select_statement` is the start symbol. A parse tree represents the sequence of rule invocations used to match an input stream, and can be constructed by *deriving* an SQL statement from the start symbol of the given SQL grammar. For each grammar rule $\alpha ::= \omega$ matched during the derivation process, the matched rule forms a branch in the parse tree, where α is the parent node, and ω represents a set of child nodes of α. A nonterminal symbol β in ω would be replaced by another grammar rule that matches the nonterminal symbol β, which in turn forms another branch originated from β. During construction of an intention tree, the placeholder meta-character represents a special type of token that can match any nonterminal and terminal symbols during derivation. In addition, lookahead on input data corresponding to a placeholder are used to distinguish alternatives. The derivation process continues recursively until all input tokens are exhausted.

In Figure 4, oval boxes represent nonterminal symbols, square boxes are terminal symbols,

and dash-lined boxes contain placeholders. In an intention tree, a placeholder is an expanding node. The branch expanded from a placeholder must follow the placeholder's intention rule. Given an intention tree, our approach uses *the grammar rule of each placeholder's parent node* as the intention rule for each placeholder. For the intention tree depicted in Figure 4, the intention rules of the three placeholders are as follows: (from left to right) two identifier lists (`id_list`), a string literal (`STRING_LIT`), and an ORDER BY clause (`order_clause`), respectively.

In addition to intention rules, the intended structure of a dynamic SQL statement includes constant symbols that are specified by developers at design-time. The intended constant symbols of an SQL statement can be represented by leaf nodes of an intention tree, excluding placeholder nodes. By walking through all leaf nodes of an intention tree, and replacing each placeholder with its intention rule, a new grammar rule can be derived for that specific dynamic SQL statement. We refer to the grammar rule derived from an intention tree as an *intention grammar*. For instance, code Fragment 2 shows the intention grammar derived from the intention tree in Figure 4, where double-quoted strings represent constant terminal symbols (e.g., "`SELECT book_name,`"), and `id_list`, `STRING_LIT`, and `order_clause` are existing grammar rules.

```
"SELECT book_name," id_list " FROM " id_list
"WHERE book_id='" STRING_LIT "' " order_clause
```
(2)

Detection of SQLIAs. Once an intention grammar is derived, an SQLIA can be detected by parsing the dynamic SQL statement using its intention grammar. If the dynamic SQL statement can be recognized by its intention grammar, then it is a benign statement; otherwise, it is malicious. For instance, while statements in both code Fragments 3 and 4 yield the same intention grammar (as shown in code Fragment 2), only the statement in Fragment 4 is malicious, as it does not conform to the intention grammar.

```
SELECT book_name, price FROM book
WHERE book_id='SQLIA' ORDER BY price
```
(3)

```
SELECT book_name, price FROM book
WHERE book_id='SQLIA'
ORDER BY price; UPDATE users SET password=null
```
(4)

Our algorithm for SQLIA detection (Algorithm 2) employs taint tracking and intention grammar derivation. The algorithm takes an SQL statement s, taint information t about s, and an SQL grammar G as arguments, and then returns a boolean to indicate whether the tainted SQL statement is malicious or not. The algorithm first constructs an intention statement s^i from an SQL statement s by replacing each consecutive tainted string in s with a meta-character. The algorithm then parses s^i using an SQL grammar G to construct an intention tree Y. Once the intention tree is constructed, the algorithm derives an intention grammar G^i by traversing through the leaf nodes of Y. If s can be parsed by G^i, the algorithm returns `false`; otherwise, it returns `true` to indicate that the intercepted SQL statement is malicious.

Intention discovery reduces the rate of false positive in the SQL detection logic. However, the intended structure expressed by a developer might allow an SQLIA to pass through. To prevent SQLIAs from a programmer's permissive intention, our "conformity to intention" heuristic employs a baseline policy to restrict where in an SQL statement the untrusted data are allowed. In our design,

Algorithm 2: IsMaliciousSQL

Input: SQL statement s
Input: s taint information t
Input: SQL grammar G
Output: A boolean value indicate whether s is malicious or not
intention statement: $s^i \leftarrow$ construct(s, t);
intention tree: $Y \leftarrow$ parse(s^i, G);
intention grammar: $G^i \leftarrow$ derive(Y);
if parse(s, G^i) failed **then**
  return true;
else
  return false;
end if

in addition to literal tokens, only identifier tokens (e.g., table name, column name) and order by, group by, and having clauses are permitted to contain tainted data.

As with all existing SQLIA detection techniques that rely on SQL grammar parsing (e.g., SQLGuard (Buehrer et al., 2005), SQLCheck (Su & Wassermann, 2006), CANDID (Bandhakavi et al., 2007)), grammatical differences between the detection engine and the back-end database could potentially cause false positives. Nevertheless, for "token type conformity", the SQL lexical analyzer in our approach is required only to be able to distinguish between literals and non-literals. Even though most database vendors develop proprietary SQL dialects (e.g., Microsoft TSQL, Oracle PL-SQL, MySQL) in addition to supporting standard ANSI SQL, the lexical analyzer required for our approach can simply treat all non-literal tokens equally and disregard the syntactical differences among SQL dialects due to different non-literal tokens supported. For instance, we used SQLPrevent with MySQL without any modification to the SQL lexer, even though the lexer was originally designed for Microsoft SQL Server. For intention discovery, we used ANSI SQL grammar during evaluation. Our implementation of SQLIA detection module can be configured to use different SQL dialects, and we are currently evaluating SQLPrevent with a real-world web application that uses Oracle as a back-end database.

Due to space limitations, we only summarize the complexity-analysis results of proposed detection logic here. For Algorithm 1, the computational complexity is $O(N)$, where N is the length of the SQL statement in characters. For Algorithm 2, the computational complexity is the same as the worst-case complexity for constructing a parse tree, which is as follows:

$$\begin{cases} O(N) & \text{if } G \text{ is LALR} \\ O(N^2) & \text{if } G \text{ is not LALR but deterministic} \\ O(N^3) & \text{if } G \text{ is non-deterministic} \end{cases}$$

Implementation

In this section, we explain the implementation of SQLPrevent in J2EE, ASP.NET, and ASP. Our description is organized around the SQLPrevent architecture depicted in Figure 2.

HTTP Request Interceptor

For J2EE, `HTTP Request Interceptor` is implemented as a servlet filter that intercepts HTTP requests. For each intercepted HTTP request, a separate instance of `TaintMark` 'wraps' the intercepted request. From this point on, on each access to the value of the request parameter, `TaintMark` calls the wrapped `HTTPServletRequest` object to get the value, marks it as tainted, and only then returns it to the caller.

Taint Tracker

The purpose of `Taint Tracker` is to mark the source of each character as either tainted or not, in an intercepted SQL statement. For J2EE, the `Taint Tracker` module is implemented as a set of *taint-enabled* classes, one for each string-related system class—such as `String`, `StringBuffer`, and `StringBuilder`. `Taint Tracker` provides dynamic per-character tracking of taint propagation in J2EE web applications. Each taint-enabled class has exactly the same class name and implements the same interfaces as the corresponding Java class—in fact, they are identical from a web application point of view. In order to specify the taintness of each character in a string, each taint-enabled class has an additional data structure referred to as taint meta-data, and a set of functions for manipulating this structure. In `Taint Tracker` for J2EE, taint meta-data is implemented as an array of booleans, with its size equal to the number of characters of the corresponding string. Each element in the array indicates whether the corresponding character is tainted or not. For taint tracking, the taint-enabled classes propagate taint meta-data during string operations. In order to replace existing system classes with `Taint Tracker` at runtime, a Java Virtual Machine (JVM) needs to be instructed to load taint-enabled classes instead of the original ones. For instance, we used the `-Xbootclasspath/p:`*`<path to taint tracker>`* option to configure Sun JVM to prepend the taint tracker library in front of the bootstrap class path.

SQL Interceptor

`SQL Interceptor` for J2EE extends P6Spy (Martin, Goke, Arvesen, & Quatro, 2003), a JDBC proxy that intercepts and logs SQL statements issued by web application programming logic before they reach the JDBC driver. JDBC is a standard database access interface for Java, and has been part of Java Standard Edition since the release of SDK 1.1. We have extended P6Spy to invoke the `SQLIA Detector` when SQL statements are intercepted.

SQL Lexer, Intention Validator and SQLIA Detector

The `SQL Lexer` module is implemented as an SQL lexical analyzer. This module converts a sequence of characters into a sequence of tokens based on a set of lexical rules, and determines the type of each token during scanning. `SQLIA Detector` takes an intercepted SQL statement as input, passes the intercepted SQL statement to the `SQL Lexer` for tokenization, then performs detection according to Algorithm 1. When a potential SQLIA is detected, `SQLIA Detector` passes the intercepted SQL statement to `Intention Validator` to check whether the query conforms to the intended syntactical structure of the designer, based on Algorithm 2. If an SQLIA is identified, the detector throws a necessary security exception to the web application, instead of letting the SQL statement through.

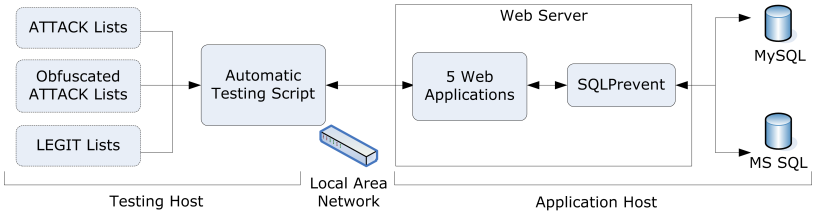

Figure 5. Design of the evaluation testbed.

Design Details Specific to ASP and ASP.NET

SQLPrevent was originally implemented in J2EE, and subsequently ported to ASP.NET and ASP in order to assess the degree to which our approach is generalizable and portable. In addition, we wanted to offer to the community a means of protecting legacy ASP applications. While the implementations of `SQL Lexer`, `Intention Validator`, and `SQLIA Detector` are identical among platforms (except the languages used), the design of `HTTP Request Interceptor`, `Taint Tracker`, and `SQL Interceptor` is specific to each execution environment. In particular, we used .NET profiling API (Pietrek, 2001) and Microsoft Intermediate Language re-writing techniques (Mikunov, 2003) to intercept SQL statements in ASP.NET. For ASP, we utilized a technique known as universal delegator (Brown, 1999) to intercept SQL statements generated from ActiveX Data Object. Due to space limitation, the design details of SQLPrevent for ASP .NET and ASP is presented in a technical report (Sun & Beznosov, 2009).

Evaluation

We evaluated SQLPrevent using the testbed suite from project AMNESIA (W. G. Halfond & Orso, 2005). We chose this testbed because it allowed us to have a common point of reference with other approaches that have used it for evaluation (W. G. Halfond & Orso, 2005; Su & Wassermann, 2006; W. G. J. Halfond et al., 2006; Bandhakavi et al., 2007; Sekar, 2009).

Experimental Setup

The experimental set up is illustrated in Figure 5. The testbed suite consisted of an automatic testing script in Perl and five web applications (Bookstore, Employee Directory, Classifieds, Events, and Portal), all included in the AMNESIA testbed. Each web application came with the ATTACK list of about 3,000 malformed inputs and the LEGIT list of over 600 legitimate inputs. In addition to the original ATTACK list, we produced another set of obfuscated attacks by obscuring the attack inputs that came with AMNESIA using hexadecimal encoding, dropping white space, and inline comments evasion techniques to validate the ability of SQLPrevent to detect obfuscated SQLIAs. To test whether the intention-validator module is capable of performing SQLIA detection without causing false positives, we modified each JSP in the testbed to intentionally include user inputs to form "`ORDER BY`" clauses in each dynamic SQL statement when an additional HTTP parameter named "`orderby`" is presented. We then modified the ATTACK and LEGIT lists by appending the additional parameter for each testing trace. To test whether the 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 10 runs.

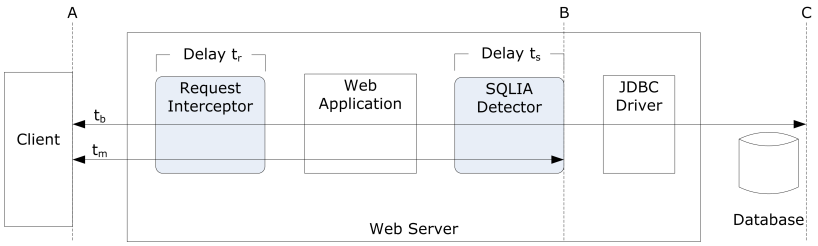

Figure 6. Detection and prevention performance evaluation. t_b and t_m are round-trip response time with SQLPrevent deployed, measured using benign and malicious requests, respectively.

Effectiveness

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.

Efficiency

We measured the performance overhead of SQLPrevent for two modes of operation: when the web application receives one request at a time, and when it is accessed concurrently by multiple web clients. First we describe the experimental setup common to both modes, then discuss specifics of experiments for each mode and the results.

To make sure the performance measurements were not skewed by hardware, we performed them on both low-end and high-end equipment. For the low-end configuration, 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. Round-trip latency, while pinging the server from the client machine, was less than 1 millisecond on average. For the high-end configuration, the testing script and web applications were installed on two identical machines, each equipped with eight Intel Xeon 2.33 GHz processors and 8 GB of memory, running Fedora Linux 2.6.24.3. Round-trip latency was less than 0.1 millisecond on average in this configuration.

Sequential Access. To measure the performance characteristics of SQLPrevent, we used nanosecond API in J2SE 1.5 and employed 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 SQLPrevent for each benign HTTP request, as shown in Figure 6, and applied the following formula: $Detection\ Overhead = (t_r + t_s)/t_b$, where t_r and t_s are the time delays for the request interceptor and SQLIA detector, respectively, and t_b is the round-trip (from A to C in Figure 6) response time when a benign SQL statement is detected.

Subject	Overhead(%)			
	Detection		Prevention	
	Avg	Std Dev	Avg	Std Dev
Bookstore	1.2	0.6	3.4	1.1
Employee	1.7	0.7	4.3	1.5
Classifieds	1.5	0.7	3.6	1.5
Events	3.3	1.4	4.2	2.3
Portal	1.9	0.9	2.5	0.5
Average	**1.9**	**0.9**	**3.6**	**1.4**

Table 1: SQLPrevent overheads for cases of benign ("detection") and malicious ("prevention") HTTP requests.

The second set of data was for measuring *prevention overhead*, which is the overhead imposed by SQLPrevent when a malicious SQL statement is detected and 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 experiment and used the following formula to calculate *prevention overhead*: $Prevention\ Overhead = (t_r + t_s)/t_m$, where t_r and t_s are the time delays for request interceptor and SQLIA detector, respectively, and t_m is the round-trip (from A to B) response time when a malicious SQL statement is detected and blocked.

For each web application, Table 1 shows the average *detection overhead* and *prevention overhead* each with its corresponding standard deviation. When averaged for the five tested applications, the maximum performance overhead imposed by SQLPrevent was 3.6% (with standard deviation of 1.4%). This overhead was with respect to an average 30 milliseconds response time observed by the web client.

Concurrent Access. To test SQLPrevent performance overhead under a high volume of simultaneous accesses, we used JMeter (Apache Software Foundation, 2007), 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 SQLPrevent and applied the *prevention overhead* formula to calculate the overhead. During stress testing, SQLPrevent imposed an average 6.9% (standard deviation 1.3%) performance overhead, with respect to an average of 115 milliseconds response time for all five applications and both databases.

Discussion

In our evaluation, SQLPrevent produced no false positives or false negatives, and imposed low runtime overhead on the testbed applications. In addition to high detection accuracy and low performance overhead, the advantages of our technique are in its automatic adaptability to developer's intentions, and its ease of integration with existing web applications.

SQLPrevent can be easily integrated with existing web applications. For instance, in order to protect a J2EE application with SQLPrevent, the administrator needs to (1) deploy the SQLPrevent

Java library into the J2EE application server, (2) configure the `HTTP Request Interceptor` filter entry in the `web.xml`, (3) replace the class name of the real JDBC driver with the class name of `SQL Interceptor` in the configuration settings, and (4) configure the JVM to prepend the `Taint Tracker` library in front of the bootstrap class path. For ASP.NET and ASP, deploying SQLPrevent is a matter of copying and registering the binary components.

We ported SQLIntention to ASP.NET and ASP to assess the generalizability of our approach, and to offer protection for legacy web applications. Legacy web applications are natural targets of SQLIAs, since most vulnerabilities are known by attackers, and the resources for prevention and protection required from development or administration might have been re-allocated to other projects. To the best of our knowledge, none of the existing dynamic SQLIA detection techniques have been ported to ASP. The lack of support for ASP is mainly due to the lack of a standard mechanism for intercepting SQL statements in ASP. Furthermore, the ASP runtime environment cannot be modified. ASP web applications have been the target of waves of massive SQLIAs from October 2007 to April 2008 (Keizer, 2008). As a consequence of these attacks, more than half a million web pages have been infected with malicious JavaScript code that redirects the visitors of compromised web sites to download malware from malicious hosts (Provos, Mavrommatis, Rajab, & Monrose, 2008). Our approach can be integrated into an existing web application with a few configuration setting changes. Security protection without additional effort from developers and administrators is vital to the protection of legacy web applications.

The approach proposed in this paper is not a replacement for all other defences against SQLIAs; it offers an alternative point in the trade-off space. Open-source and some other applications—source code for which can be analyzed and, if necessary, modified by the application owners—make those approaches that employ static analysis and/or alteration of the source code viable. For applications where an additional overhead of 2-5% is unacceptable, static detection and elimination of SQLIA vulnerability identification, or even the use of parameterized query APIs, would be more appropriate. Our approach offers the ability to protect existing applications effectively, efficiently, and without having to depend on application vendors or developers.

The concept of token type conformity and conformity to intention can be applied to other types of web application security problem such as cross-site scripting (XSS) and remote command injection, for which taintness of tokens can be analyzed and the intended syntactical structures can be dynamically discovered. For instance, a web application can check whether tainted data is used to construct script elements in the Document Object Model (DOM) of a dynamically generated HTML page to prevent XSS attacks.

Conclusion

SQL injection vulnerabilities are ubiquitous and dangerous, yet many 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 possibilities of false positive, 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 some real world settings. In this paper, we presented a novel approach to runtime SQLIA protection, as well as a tool (SQLPrevent) that implements our approach. Our experience and evaluation of SQLIntention indicate that it is effective, efficient, 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 apply dynamic intention discovery to prevent other types of web application attacks, and to port our approach to PHP in order to provide protection to web applications developed in this popular platform. To obtain more realistic data on the practical possibility of false positives and false negatives, we plan to evaluate SQLPrevent on real-world web applications, and make SQLPrevent an open source project. We also plan to apply SQLPrevent to dynamic discovery of SQLIA vulnerabilities.

References

Anley, C. (2002). Advanced SQL injection in SQL server application. *Technical report, NGSSoftware Insight Security Research (NISR)*. http://www.nextgenss.com/papers/advanced_sql_injection.pdf.

Apache Software Foundation. (2007). *Apache JMeter.* http://jakarta.apache.org/jmeter/.

AQTRONIX. (2007). *WebKnight.* http://www.aqtronix.com/?PageID=99.

Bandhakavi, S., Bisht, P., Madhusudan, P., & Venkatakrishnan, V. N. (2007, October). CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on computer and communications security (CCS)* (pp. 12–24). Alexandria, Virginia, USA.

Boyd, S. W., & Keromytis, A. D. (2004, June). SQLrand: Preventing SQL injection attacks. In *Proceedings of the second international conference on applied cryptography and network security (ACNS)* (pp. 292–302).

Breach Security Inc. (2007). *ModSecurity.* http://www.modsecurity.org/.

Brown, K. (1999, January). Building a lightweight COM interception framework part 1: The universal delegator. *Microsoft Systems Journal*.

Buehrer, G. T., Weide, B. W., & Sivilotti, P. A. G. (2005, September). SQLGuard: Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th international workshop on software engineering and middleware* (pp. 106–113). Lisbon, Portugal.

Halfond, W. G., & Orso, A. (2005). AMNESIA: Analysis and monitoring for neutralizing SQL injection attacks. In *Proceedings of the 20th IEEE/ACM international conference on automated software engineering* (pp. 174–183). Long Beach, California, USA.

Halfond, W. G., Viegas, J., & Orso, A. (2006). A classification of SQL injection attacks and countermeasures. In *IEEE international symposium on secure software engineering.*

Halfond, W. G. J., Orso, A., & Manolios, P. (2006). Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on foundations of software engineering* (pp. 175–185). Portland, Oregon, USA.

Howard, M., & LeBlanc, D. (2003). *Writing secure code* (2nd ed.). Redmond, Washington: Microsoft Press.

Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D. T., & Kuo, S.-Y. (2004). Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web* (pp. 40–52).

Jovanovic, N., Kruegel, C., & Kirda, E. (2006, May). Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE symposium on security and privacy.*

Julisch, K., & Darcier, M. (2002). Mining intrusion detection alarms for actionable knowledge. In *Proceedings of the 8th ACM international conference on knowledge discovery and data mining* (p. 366-375).

Keizer, G. (2008, April). Huge web hack attack infects 500,000 pages. *Computerworld*. http://www.computerworld.com/action/article.do? command=viewArticleBasic&articleId=9080580.

Livshits, V. B., & Lam, M. S. (2005, August). Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th USENIX security symposium* (pp. 271–286).

Maor, O., & Shulman, A. (2005). SQL injection signatures evasion. *White Paper of Imperva Inc.*. http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html.

Martin, A., Goke, J., Arvesen, A., & Quatro, F. (2003). *P6Spy open source software.* http://www.p6spy.com/.

Mikunov, A. (2003, September). Rewrite MSIL code on the fly with the .NET framework profiling API. *Microsoft MSDN Magazine*. http://msdn.microsoft.com/en-us/magazine/cc188743.aspx.

Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., & Evans, D. (2005, May 30 - June 1). Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP international information security conference* (pp. 296–307). Makuhari-Messe, Chiba, Japan.

OWASP. (2008). *Open web application security project (OWASP) top ten project.* http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

Pietraszek, T., & Berghe, C. V. (2005). Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th international symposium on recent advances in intrusion detection* (pp. 124–145).

Pietrek, M. (2001, December). The .NET profiling API and the DNProfiler tool. *Microsoft MSDN Magazine*. http://msdn.microsoft.com/en-us/magazine/cc301725.aspx.

Provos, N., Mavrommatis, P., Rajab, M. A., & Monrose, F. (2008, June 22-27). All your iFRAMEs point to us. In *Proceedings of 17th USENIX security symposium.* Boston, Massachusetts, USA.

Scott, D., & Sharp, R. (2002, May). Abstracting application-level web security. In *Proceedings of the 11th international conference on the World Wide Web* (pp. 396–407). Honolulu, Hawaii, USA.

Sekar, R. (2009, February 8–11). An efficient black-box technique for defeating web application attacks. In *Proceedings of the 16th annual network and distributed system security symposium NDSS'09.* San Diego, CA, USA.

Su, Z., & Wassermann, G. (2006, January). The essence of command injection attacks in web applications. In *Proceedings of the 33rd annual ACM SIGPLAN - SIGACT symposium on principles of programming languages* (pp. 372–382). Charleston, South Carolina, USA.

Sun, S.-T., & Beznosov, K. (2009, March 30). *SQLPrevent: Effective dynamic detection and prevention of SQL injection attacks* (Technical Report No. LERSSE-TR-2009-032). Laboratory for Education and Research in Secure Systems Engineering, University of British Columbia. http://lersse-dl.ece.ubc.ca.

Valeur, F., Mutz, D., & Vigna, G. (2005). A learning-based approach to the detection of SQL attacks. In *Proceedings of the conference on detection of intrusions and malware & vulnerability assessment (DIMVA 2005)* (pp. 123–140).

Wall, L. (2007). *perlsec - perl security* (Library No. v.5.10). perl.org. http://perldoc.perl.org/perlsec.html.

Werlinger, R., Hawkey, K., Muldner, K., Jaferian, P., & Beznosov, K. (2008, July 23-25). The challenges of using an intrusion detection system: Is it worth the effort? In *Proceedings of the 4th symposium on usable privacy and security (SOUPS)* (p. 107-116). Pittsburgh, PA.

Xie, Y., & Aiken, A. (2006, August). Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX security symposium* (pp. 179–192).

Zetter, K. (2009, March 24). 'The Analyzer' hack probe widens; $10 million allegedly stolen from U.S. banks. *Wired Magazine*. http://blog.wired.com/27bstroke6/2009/03/the-analyzer-ha.html.