SQLPrevent: Effective dynamic detection and prevention of SQL injection

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

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

Technical report LERSSE-TR-2009-032[‡]

Last Modification Date: 2009/03/30

Revision: #14

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

Abstract

This paper presents an approach for retrofitting existing web applications with run-time protection against known as well as unseen SQL injection attacks (SQLIAs). This approach (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 require modification of the runtime environment, such as PHP interpreter or JVM, and (4) is independent of the back-end database used. The approach precision 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.

We have implemented the proposed approach in the form of protection mechanisms for J2EE applications. Named SQLPrevent, these mechanisms intercept both HTTP requests and SQL statements, mark and track parameter values originated from HTTP requests, and perform SQLIA detection and prevention on the intercepted SQL statements. We extended the AMNESIA testbed to contain false positive testing traces, and employed the extended testbed to evaluate SQLPrevent over 15,000 unique HTTP requests with five web applications. In our experiments, SQLPrevent produced no known false positives or false negatives, and imposed a 3.6% performance overhead with respect to 30 millisecond response time in the tested applications. We also ported SQLPrevent to ASP.NET and ASP, which is of vital importance to the protection of legacy ASP applications, as they have been the target of several massive SQLIAs since October 2007.

Contents

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 2008 [OWA08]. Since they are easy to find and exploit, SQL injection vulnerabilities are frequently exploited by attackers. As a case in point, SQLIAs were apparently employed by Ehud Tenenbaum, who has been arrested on charges of stealing $1.5M from Canadian and at least $10M from US banks [Zet09]. An effective and easy to employ method for protecting numerous existing web applications from SQLIAs is crucial for the security of today's organizations.

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 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 being effective [Anl02a, Anl02b, Cer03] and many web applications deployed today are still vulnerable to SQLIAs [MIT08]. The reasons are manifold:

- 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, Bre07] are implemented as proxy servers to prevent malicious input reaching vulnerable web applications. They can be deployed without modifying the existing web applications. However, these tools suffer from both false positives and false negatives [HVO06]. An accuracy of 100% is claimed by some techniques that combine design-time static analysis/arugment and runtime protection [BK04, HO05, BWS05, SW06, BBMV07]. However, they require application code to be analyzed/instrumented or manually modified. Source code analysis, instrumentation, or manual modification are especially problematic for small organizations and legacy web applications, where source code, qualified developers, or security development process might not be available. Approaches employing dynamic taint analysis [NTGG$^+$05, PB05] use HTTP requests and SQL statements for SQLIA detection, which do not require access to the application source code. However, these approaches require modification of runtime environment (i.e., PHP interpreter), which may be impractical for other web execution platforms, such as Java, ASP.NET, or ASP.

Moreover, a common deficiency of existing SQLIA approaches based on analyzing dynamic SQL statements [BK04, HO05, BWS05, NTGG$^+$05, PB05, SW06, BBMV07] is in defining SQLIAs too restrictively, which leads to a larger than necessary percentage of false positives (FPs). False positives could have significant negative impact on the utility of detection and protection mechanisms, because investigating FPs takes time and resources [JD02, WHM$^+$08]. Even worse, if the rate of FPs is high, security practitioners might get conditioned to ignore them.

In this paper, we propose an approach for retrofitting existing web applications with run-time protection against known as well as unseen SQL injection attacks (SQLIAs). 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 [HO05] testbed, which has been used for evaluating several other research systems [HO05, 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. We extended AMNESIA testbed to contain requests with new false positives, and also added 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 (average 3.6%, standard deviation 1.4%) with respect to 30 milliseconds response time in the tested 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. SQLPrevent can be employed as a stand-alone solution or integrated with existing SQLIA detection techniques to reduce the possibility of false positives.

ASP web applications have been the target of waves of massive SQLIAs since October, 2007 [Lan08, Kei08, Lem08]. As the consequence of those attacks, more than half a million web pages have been implanted with malicious JavaScript code that redirects the visitors of compromised web sites to the distributors of malware [PMRM08]. And yet, none of the existing dynamic SQLIA detection techniques [BK04, HO05, BWS05, NTGG$^+$05, PB05, SW06, BBMV07] has been ported to ASP. We ported SQLPrevent to ASP.NET and ASP to demonstrate the generalizabilty of our approach, and to protect legacy web 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. Afterwards, we describe our approach for detecting and preventing SQL injection attacks. Then, we explain the implementation of SQLPrevent in J2EE, ASP.NET, and ASP, followed by a description of the evaluation methodology and results. Afterwards, we discuss the implications of the results and the strengths and limitations of our approach. In the

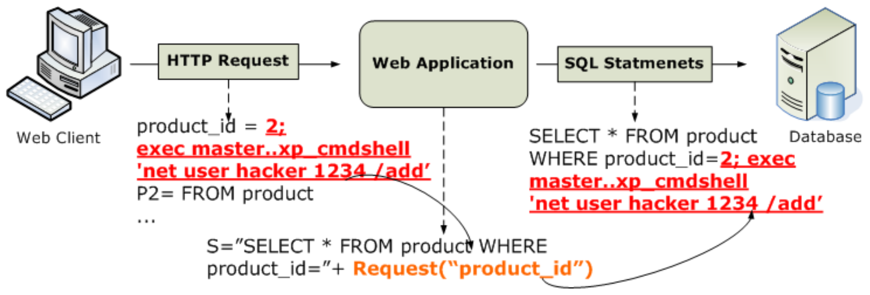


Figure 1: How SQL injection attacks work.

last section, we summarize the paper and outline future work.

2 Background

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

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, as illustrated in Figure 1. 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 that creates a new user account in the Window operating system by appending the attack string “`exec master..xp_cmdshell 'net user hacker 1234 /add'`” to the legitimate input `product_id=2`, as shown in the following fragment:

```
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 [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),0x6578656
320206D61737465722E2E78705F636
D647368656C6C20276E6574207573
6572206861636B6572
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 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 those 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.

Example 1 *We illustrate this false positive problem through a running example. Assume there is an HTML dropdown list named "order_by", which consist 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 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.

2.3 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 [AQT07] and ModSecurity [Bre07] 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 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 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.

3 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. 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 the above categorization, and briefly summarize the advantages and limitations of existing approaches. 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 [HGM04], and approaches proposed by Livshits et al. [LL05], Jovanovic et al. [JKK06], 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 cannot be guaranteed. Non-perfect training data causes such techniques to produce false positives and false negatives. Our approach does not rely on the ability of the application developers or owners to acquire a qualified "perfect" data set—which has all possible versions of legitimate SQL statements and yet has no SQLIAs.

Static analysis with runtime protection: Approaches in this category combine design-time source code analysis/instrumentation/augmentation with dynamic runtime protection. These approaches identify the intended structures of SQL statements at development time, and check 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. For our running Example 1 of false positive, the intercepted SQL statement in SQLrand would be as "`SELECT`^{key} `c1 FROM`^{key} `t1` <u>`ORDER BY id`</u>", where "`key`" represents the random key. The intercepted SQL statement would cause a false positive since keywords "`ORDER`" and "`BY`" are not appended with the random key.

SQLGuard [BWS05] provides programmers with a Java library to manually bracket the placeholders of user input in SQL statements. The library also contains an implementation of proxy JDBC driver to be used in place of the original one. During runtime, before passing a query to the delegated JDBC driver for execution, SQLGuard compares two parse trees of the dynamically created SQL statement with and without input values respectively. If the structures of two parse trees are identical, the query is considered benign, otherwise, it is malicious. In the case of Example 1, SQLGuard will compare two parse trees of (1) "`SELECT c1 FROM t1` *key*<u>`ORDER BY id`</u>*key*", and (2) "`SELECT c1 FROM t1` *key key*", where the first query contains input value whereas the second does not. SQLGuard would trigger an alarm for this query since both augmented queries are not valid SQL statements.

AMNESIA [HO05] builds legitimate SQL statement models using static analysis based on information-flow. Each model is a non-deterministic finite-state automaton, in which the transition labels are represented by SQL tokens with placeholders for literal values. At each database access point in the application, AMNESIA instruments calls to the runtime monitor. 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 "$\texttt{SELECT} \rightarrow \texttt{c1} \rightarrow \texttt{FROM} \rightarrow \texttt{t1} \rightarrow \beta$" 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 [HOM06] prevents SQLIAs by checking whether all SQL keywords and operators in an SQL statement are marked as trusted. If there is any character in those keywords and operators not marked as trusted, the SQL statement is rejected. 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.

The instrumentation also replaces Java string-related classes in the bye-code with classes from a specialized Java library developed by the authors to track taint propagation, and adds statements to JDBC calls to intercept SQL statements before submitting to back-end database. The main limitations of WASP are suboptimal reliability, efficiency, completeness, and correctness. First, intrusive instrumentation of byte code can affect the stability and robustness of the target application, leading to its reduced reliability. Second, WASP conservatively marks and keeps track of all hard-coded, implicitly-created strings in the source code as trusted. However, most of those strings are not used in the construction of SQL statements, introducing unnecessary performance overhead. Third, the coverage of implicitly-created strings by Java compiler is not guaranteed. It is possible that there are some implicitly-created strings introduced by a particular version/implementation of Java compiler which are not covered by WASP. For instance, some Java compilers use StringBuilder while other use StringBuffer when string concatenation (i.e., +) is encountered in the source code. Last but not least, WASP is prone to false positives. 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 [SW06] detects SQLIAs by observing the syntactic structure of generated SQL queries and checking whether this syntactic structure conforms to an augmented grammar. In SQLCheck, an augmented grammar is a standard SQL grammar instrumented with a security policy. The security policy specifies which symbol (either terminal or non-terminal) in the grammar is permitted to contain user inputs. Each permitted symbol is paired with a special meta-character, and then added to the grammar as an alternative rule to the existing symbol. The main limitation of SQLCheck is that it requires that each parameter value get 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 a web application displays a parameter value such as current login user directly in the response page, the user name would be prepended and appended with special meta-character. Or 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 be 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 [BBMV07] 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 of 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. If two parse trees are not identical, the query is considered malicious. 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 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 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 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.

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, ASP.NET, or ASP, and (2) need all database access functions to be identified in advance. Our approach has neither limitation.

Sekar [Sek09] 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.

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%), as reported by the approach authors, and the need for application developers to be involved in the SQLIA vulnerability testing.

When a keyword token is tainted in a dynamic SQL statement, all the approaches discussed above except SQLCheck would trigger an SQLIA alarm. SQLCheck determines where user input is permitted in an SQL statement based on an augmented SQL grammar. However, the augmented SQL grammar might not match all intended syntactical structures in the programming logic. Another 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 query. 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 a web application displays a wrapped parameter value such as current login user directly in the response page, the user name would be pre- and post-pended with special meta-characters. 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).

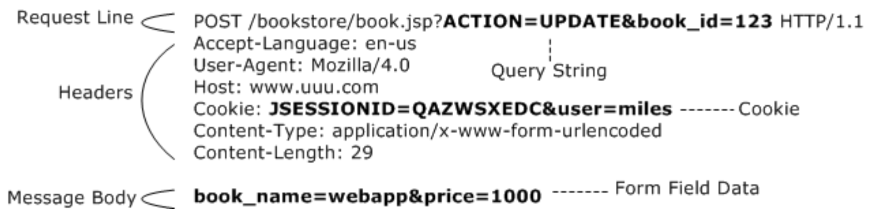
In addition to false positives, some existing approaches (e.g., SQLGuard, SQLrand, SQLCheck) require access to the application source code. Such access is sometimes not possible (as in the case of including calls to a binary component provided by a third-party), and repeated analysis or modification increases the overhead of change management. Source code analysis, instrumentation, or manual modification is especially problematic for small organizations and legacy web applications, where source code, qualified developers, or security development processes might not be available.

4 Approach

Our approach enables retrofitting existing web applications with run-time protection against known as well as unseen SQL injection attacks (SQLIAs). Named SQLPrevent, the proposed run-time protection mechanism (1) intercepts HTTP requests and SQL statements at runtime, (2) marks parameter values in HTTP request as tainted, (3) tracks taint propagation during string manipulations, and (4) performs analysis on the intercepted SQL statements with the assistance of taint information. Our SQLIA detection logic can be configured to employ either one of the two proposed heuristics or both.

Referred as *strict taint policy*, the first heuristic triggers an alarm if the content of the corresponding HTTP request's parameters is used in non-literal tokens (e.g., identifiers or operators) of the SQL statement. While highly efficient (with less than 3% of overhead), this heuristic leaves room for false positives when SQL keywords or operators are intentionally included in a dynamic SQL statement, triggering an SQLIA alarm, even though the query does not result in an SQLIA.

Referred as *intention conformation*, our second heuristic aims at eliminating the above type of false positives in the SQLIA detection logic. The heuristic enables runtime discovery of the developers' intention for individual SQL statements made by web applications. 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 an SQL grammar, which we named *intention grammar*. Our detection algorithm triggers an alarm if the intercepted SQL statement does not conform to the corresponding intention grammar. Depending on



```
POST /bookstore/book.jsp?ACTION=UPDATE&book_id=123 HTTP/1.1
Accept-Language: en-us
User-Agent: Mozilla/4.0
Host: www.uuu.com
Cookie: JSESSIONID=QAZWSXEDC&user=miles
Content-Type: application/x-www-form-urlencoded
Content-Length: 29

book_name=webapp&price=1000
```

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

the configuration of SQLPrevent, the alarm can be just sent to the audit log or an IDS monitor, or the corresponding SQL query can be prevented from execution. The following subsections describe each of the heuristics.

4.1 Strict Taint Policy

For the purpose of explaining our approach, we abstract a web application as a function that takes HTTP requests as inputs and generates SQL statements as outputs. We exclude from this abstraction communications made by web applications to other data sources such as XML documents, LDAP and other directory servers, or inputs or outputs from/to files. 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 (1) request line with optional query strings, (2) headers, and (3) message body, as illustrated in Figure 2. We abstract an HTTP request in the context of SQLIAs as a *set of name-value pairs* in which the name part serves as an 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. As an example, Table 4.1 shows an abstraction of the HTTP request from 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

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

4.1.1 Alteration of the SQL Statement Syntactical Structure by SQLIAs

The core of the strict taint policy heuristic is in an 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. A string or numeric literal represents a fixed string or number value within an SQL statement. For example, in the statement "`UPDATE books set book_name='SQLIA', price=1000 WHERE book_id=123`", "`SQLIA`" is a string literal enclosed by a pair of single quotes, and "`1000`" and "`123`" are both numeric 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 intended placeholders for the parameter values. When the placeholders are instantiated with parameter values, those values should only be used as string or numeric literals in order to maintain the original syntactical structure of the SQL statement expressed by the application developers. Otherwise, an adversary can inject extra single quotes, SQL keywords, operators, or delimiters into the SQL statements to alter its syntactical structure.

4.1.2 Tracking of Tainted Data

Tainted data refers to data which originates from an untrusted source, such as an HTTP request. An SQLIA occurs when tainted data is used to construct an SQL statement in a way that alters the intended syntactical structure of the SQL statement. In the case of strict taint policy, tainted data should be used only as string or numeric literals to construct SQL statements. Thus, while "`UPDATE books SET book_name='`<u>`SQLIA`</u>`', price=`<u>`1000`</u>` WHERE book_id=`<u>`123`</u>" is a benign SQL statement (underlined labels indicate the data is tainted), "`UPDATE books SET book_name='SQLIA', price=1000 WHERE book_id=`<u>`123 SHUTDOWN`</u>" is not. In the latter case, tainted data <u>`123 SHUTDOWN`</u> are used as a numeric value and an SQL command as well.

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.	1000	[LITERAL - INTEGER]
11.	WHERE	[IDENTIFIER]
12.	book_id	[IDENTIFIER]
13.	=	[OPERATOR - EQUALS]
14.	123	[LITERAL - INTEGER]

Table 2: Tokens and their types generated by an SQL lexer after performing lexical analysis on statement "`UPDATE books SET book_name='SQLIA', price=1000 WHERE book_id=123`".

To trace the source of each character in an SQL statement for J2EE web applications, we implemented per-character taint propagation using custom implementation of Java's string-related classes. Our implementation (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. For instance, consider `s1="Hi Taint"` and `s2="Sun"`, and assume `s3=s1.substring(0,5)+s2`. After extracting substring from `s1` and concatenating with `s2`, the resulted string `s3` is `Hi TaiSun`, which contains both corresponding taint meta-data from `s1` and `s2` during string operations.

Our implementation of taint tracking module can be integrated into existing J2EE web applications without modifications to either runtime or applications. A Java Virtual Machine (JVM) can be configured to load taint-enabled classes instead of the original runtime library by prepending taint tracking library in front of bootstrap class path. For instance, `bootclasspath` option of Sun JVM could be used for this purpose.

For ASP.NET and ASP, our implementation of taint tracking module retrieves the corresponding HTTP request from current thread-local storage (ASP version gets the corresponding request object from the context of Microsoft Transaction Server), and performs string comparison with the intercepted SQL statement. For each parameter value in the stored HTTP request, if the input value appears as a substring of the SQL statement, taint tracking module marks the corresponding substring in the SQL statement as tainted.

4.1.3 Lexical Analysis of SQL Statements

We perform lexical analysis of SQL statements at run-time in order to identify those tokens in the statements that are not string or number literals. This is necessary for checking whether any character in those tokens is tainted. Because the exact types of non-literal tokens do not have to be identified, the implementation can be efficient and work for most flavors of SQL. Even when a particular database has a different syntax for non-literals, re-implementing such a coarse-grain lexical analyzer is simpler than a complete analyzer as used by database servers or other protection mechanisms that utilize an SQL parser. 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.

Lexical analysis is the process of generating a stream of tokens from the sequence of input characters comprising the SQL statement. An SQL lexer performs a simple left-to-right scan of the input and groups characters that have a cohesive, collective meaning. These groups of characters are called *tokens*. Tokens include string, numbers, identifiers, keywords, operators and miscellaneous punctuation symbols. For instance, the lexical rule of most SQL dialects—such as Microsoft T-SQL [Mic07], Oracle PL-SQL [Ora07], and MySQL [MyS07]—specifies that a string token is a sequence of characters enclosed within either single quote (') or double quote (") characters. An identifier token is a sequence of letters and digits, and a number token is a sequence of digits. Given an SQL statement, an 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=1000 WHERE book_id=123`", an SQL lexer will generate the set of tokens and the corresponding token types shown in Table 2.

The goal of lexical analysis in our approach is to generate two set 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. Based on our design, the exact types of tokens in the NON-LITERAL set are irrelevant for the purpose of our detection logic. This is why `UPDATE`, `SET`, and `WHERE` are classified as *identifiers* in Table 2, although they are actually SQL *keywords*. This simplified design of the lexical analyzer makes our approach efficient and more portable among databases.

4.1.4 Detecting SQLIAs

Applying our conjecture 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 strict taint policy. 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., 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 malicious.

Algorithm 1: Strict taint policy SQLIA detection algorithm

```
Input: An intercepted SQL statement string $s$
Output: A boolean value indicate 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;
```

To analyze the computational complexity of Algorithm 1, let N be the length of the SQL statement in characters, and M be the number of tokens. The detection algorithm performs lexical analysis by going through N characters in the SQL statement and parsing it into a set of tokens. The complexity of this step is $O(N)$. If we assume that token type is determined during lexical analysis then `typeOf(`t`)` takes $O(1)$. For each token t, `isTainted(t)` goes through each character of t in the taint meta-data to determine whether any character in it is tainted. Since the length of the token's taint meta-data is linearly proportional to the length of the token, the complexity of `isTainted(t)` is $O(N/M)$. Thus, the overall computational complexity of the detection algorithm is $O(N) + M * O(N/M) = O(N)$.

The "token type conformity" heuristic was originally inspired by Perl taint mode [Wal07]. 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. [NTGG$^+$05] and [PB05] 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

```
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).

should validate any binary data retrieved from an unsafe buffer [HL03].

4.2 Intention Conformation

To protect the integrity of SQL statements, our strict taint policy heuristic and some other approaches use pre-defined taint policies, implicitly or explicitly, to specify where in an SQL statement the untrusted data is 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. While these approaches are effective, however, 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 one step further, and treat SQLIA as an instance of the *intention conformation* problem. Our second heuristic 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. When SQLPrevent is configured to employ the heuristic of intention conformation, it (1) intercepts HTTP requests and SQL statements, (2) performs dynamic taint tracking, (3) encodes the intended syntactical structure in the form of an SQL grammar, which we named *intention grammar*, and (4) validates a dynamically constructed statement against its intention grammar. The last two steps are new, compared to the first heuristic. In what follows, we explain our heuristic by discussing departing observations, intuition, and technical details.

4.2.1 Intention Statement

Web application developers typically specify 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 HTTPRequest 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 SQL statement "`SELECT book_name,price FROM book WHERE book_id='SQLIA' ORDER BY price`" is intercepted, our approach substitutes each tainted substring with 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' 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 is served as the intended syntactical structure, and can be discovered by using an SQL parse tree.

4.2.2 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 SQL parse tree. Our approach constructs a parse tree (referred 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 a 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

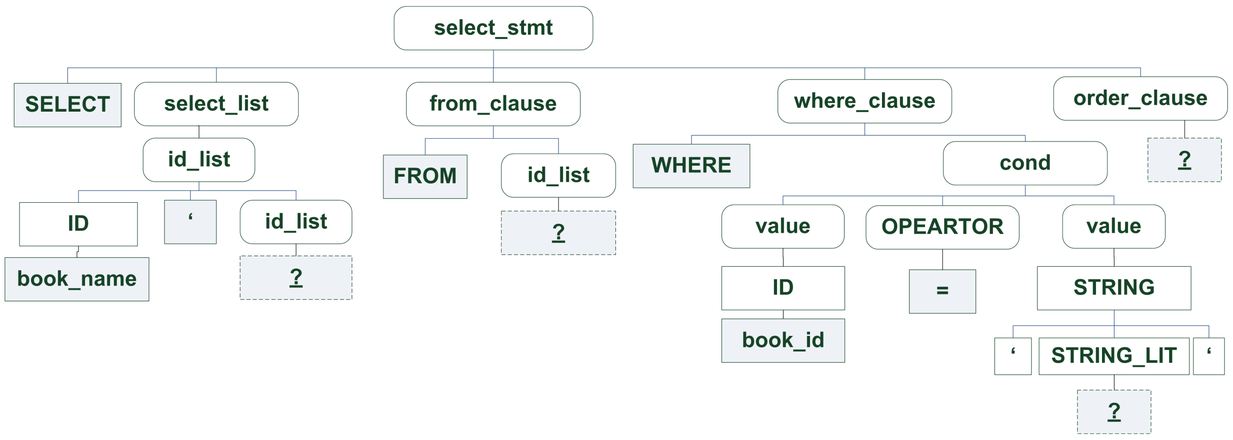


Figure 4: The intention tree of the intention statement from Fragment 1. Oval boxes represent *nonterminal symbols*, square boxes represent *terminal symbols*, and dash-lined boxes are placeholders. The grammar rules for each placeholder are (from left to right) two `id_lists`, a `STRING_LIT`, and an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

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 which can match any nonterminal and terminal symbols during derivation, and 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 example intention tree in Figure 4, the intention rules of each placeholder 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 also 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

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: $\Upsilon \leftarrow$ parse($s^i$, $G$);
intention grammar: $G^i \leftarrow$ derive($\Upsilon$);
if parse($s$, $G^i$) failed then
  return true;
else
  return false;
end if
```

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.

$$\begin{array}{l}\texttt{"SELECT book\_name," id\_list " FROM " id\_list} \\ \texttt{"WHERE book\_id='" STRING\_LIT "' " order\_clause}\end{array} \tag{2}$$

4.2.3 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.

$$\begin{array}{l}\texttt{SELECT book\_name, }\underline{\texttt{price}}\texttt{ FROM }\underline{\texttt{book}} \\ \texttt{WHERE book\_id='}\underline{\texttt{SQLIA}}\texttt{' }\underline{\texttt{ORDER BY price}}\end{array} \tag{3}$$

$$\begin{array}{l}\texttt{SELECT book\_name, }\underline{\texttt{price}}\texttt{ FROM }\underline{\texttt{book}} \\ \texttt{WHERE book\_id='}\underline{\texttt{SQLIA}}\texttt{'} \\ \underline{\texttt{ORDER BY price; UPDATE users SET password=null}}\end{array} \tag{4}$$

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 returns a boolean indicating 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 Υ. Once the intention tree is constructed, the algorithm then derives an intention grammar G^i by traversing through the leaf nodes of Υ. If s can be parsed by G^i, the algorithm returns `false`, otherwise it returns `true` to indicate the intercepted SQL statement is malicious.

To analyze the computational complexity of Algorithm 2, let N be the length of s in characters and M be the number of tokens in it. Replacing each tainted substring with placeholder character requires going through N characters of s, which is $O(N)$. Constructing intention tree Υ requires parsing s^i. According to Aho et al. [ALSU07], the worst-case complexity for constructing a parse tree is as follows:

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

An intention grammar can be derived by walking through the leaf nodes of a parse tree, which is $O(M)$. Given that $M \leq N$, the overall computational complexity of the detection algorithm same as for constructing a parse tree for s using grammar G, which is shown above. 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, 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 [BWS05], SQLCheck [SW06], CANDID [BBMV07]), 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.

5 Implementation

To evaluate our approach, we implemented Algorithms 1 and 2 in a protection mechanism referred in this paper as SQLPrevent. It intercepts HTTP requests, tracks propagation of tainted data, determines if an SQL statement is malicious, and if so, throws an exception, instead of passing the statement to the database. As illustrated in Figure 5, SQLPrevent consists of `HTTP Request Interceptor`, `Taint Tracker`,

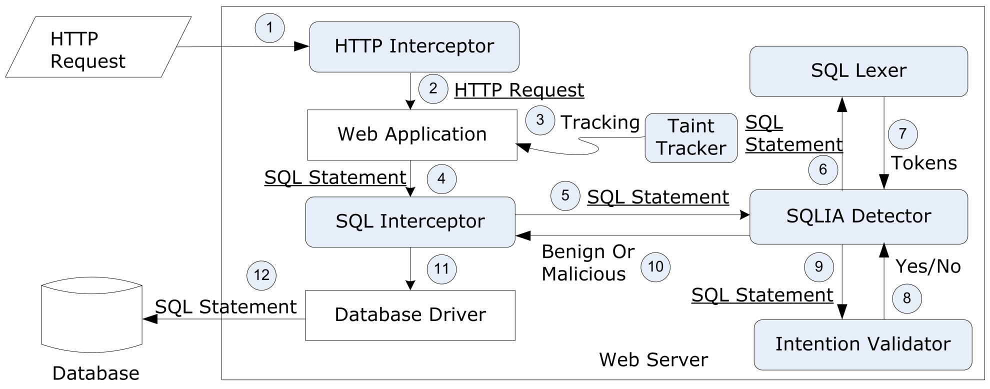


Figure 5: Main elements of SQLPrevent architecture are shown in light grey. The data flow is depicted with sequence numbers and arrow labels. Underlined labels indicate that the data is accompanied by the taint meta-data. Depends on runtime configuration, data may flow to SQL Lexer or Intention Validator accordingly.

`SQL Interceptor`, `SQL Lexer`, `Intention Validator`, and `SQLIA Detector` modules. Each module is described in the following subsections. We first explain the design of the Java version of SQLPrevent and then describe specifics of ASP and ASP.NET versions.

5.1 HTTP Request Interceptor

For J2EE, `HTTP Request Interceptor` is implemented as a servlet filter [Cow01] 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 wrapped `HTTPServletRequest` object to get the value, marks it as tainted, and only then returns it to the caller. Note that since the SQLPrevent implementation uses standard J2EE interfaces, no changes to the web application are required in order for the `TaintMark` to wrap the original request object.

5.2 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, `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 same class name and implements same interfaces as the corresponding Java class—in fact, they are identical from a web application point of view. In order to specify taintness of each character in a string, each taint-enabled class has an additional data structure referred 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 following string operations: (1) string concatenation, (2) return from a function, (3) string conversions, such as `toUpperCase()` and `trim()`, and (4) string extraction, such as `substring()`. 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 `-Xbootclasspath/p:`*`<path to taint tracker>`* option to configure Sun JVM to prepend taint tracker library in front of bootstrap class path.

5.3 SQL Interceptor

`SQL Interceptor` for J2EE extends P6Spy [MGAQ03], 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.

5.4 SQL Lexer, Intention Validator and SQLIA Detector

`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. For strict taint policy approach, `SQLIA Detector` takes an intercepted SQL statement as input, and passes the intercepted SQL statement to the `SQL Lexer` for tokenization, and then performs detection according to Algorithm 1. For intention conformation approach, `SQLIA Detector` passes the intercepted SQL statement directly to `Intention Validator`. `Intention Validator` checks whether the query conforms to the intended syntactical structure of designer based on Algorithm 2. 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.

As illustrated in Figure 5, the original data flow (HTTP request $\rightarrow$ web application $\rightarrow$ database driver $\rightarrow$ database) is modified when SQLPrevent is deployed into a web server. First, the reference to the program object representing an incoming HTTP request is intercepted. Second, the SQL statements composed by web applications are intercepted by the `SQL Interceptor` and passed to the `SQLIA Detector`. The latter then calls `SQL Lexer` to detect potential SQLIAs and then passes to `Intention Validator` module to check it is an instance of false positive. If the intercepted SQL statement is malicious, the SQL interceptor prevents the malformed SQL statement from being submitted to the database.

5.5 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 how much 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. This subsection provides ASP(.NET)-specific design details of these three modules of SQLPrevent.

5.5.1 HTTP Request Interceptor

`HTTP Request Interceptor` for ASP.NET is implemented as an HTTP module [Mic08a]—a component type introduced in ASP.NET 1.0. An HTTP module is a .NET component that implements the `System.Web.IHttpModule` interface. HTTP modules can be integrated into the ASP.NET request processing pipeline by registering certain events of interest. Whenever a registered event occurs, ASP.NET invokes the interested HTTP modules to interact with the request. The implementation of `HTTP Request Interceptor` handles `BeginRequest` event. This event handler 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 `Taint Tracker` module when it processes the intercepted SQL statement. 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 multithreaded 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 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. ASP does not provide a standard way to intercept HTTP requests. However, Internet Information Server (IIS) automatically stores ASP intrinsic objects (including "`Request`" object) in the context of Microsoft Transaction Server (MTS). MTS is a required component in the execution environment of ASP. ASP version of `Taint Tracker` module retrieves the corresponding HTTP request instance by calling `GetObjectContext` function provided by MTS.

5.5.2 Taint Tracker

ASP.NET and ASP have no standard mechanisms for replacing system string-related classes as in J2EE. Therefore, in order to trace the source of each character in an intercepted SQL statement, we implemented `Taint Tracker` for ASP.NET to retrieve the corresponding HTTP request from current thread-local storage (ASP version gets the corresponding request object from MTS), and to perform string comparison with the intercepted SQL statement to mark tainted characters.

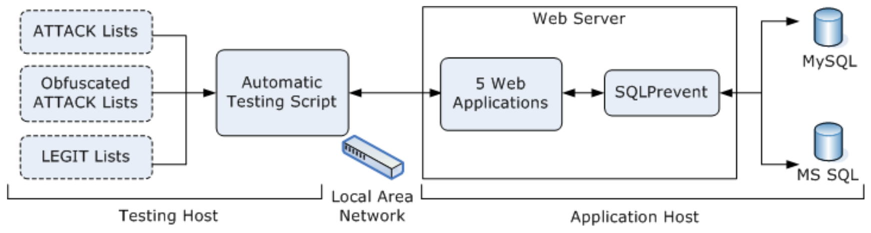


Figure 6: Design of the evaluation testbed.

5.5.3 SQL Interceptor

Since ASP.NET has no open mechanism for intercepting SQL statements, we used .NET profiling API [Pie01] and Microsoft Intermediate Language (MSIL) re-writing techniques [Mik03] to intercept SQL statements. Microsoft .NET profiling API is a standard mechanism for application or tool developers to instrument the behavior of a .NET program by providing components that register events of interest, such as just-in-time compilation or method invocation. Right before a .NET *managed* method is invoked, Common Language Runtime (CLR) loads the .NET assembly that contains the method code into memory, and compiles it into executable by the target CPU (a process called just-in-time compilation). ASP.NET version of `SQL Interceptor` provides components that register just-in-time compilation events of database-access classes such as `SQLCommand` and `OleDbCommand`. In the event handler, `SQL Interceptor` employs MSIL re-write technique to install a hook that calls `SQLIA Detector` module in the beginning of each method of interest. Those mechanisms are supported by all versions of .NET, and can be deployed into existing ASP.NET web application dynamically without access to application source code. ASP, executed in the domain of COM (Common Object Model) [Mic08c], has no default mechanism for intercepting SQL statements, either. In COM, ActiveX Data Object (ADO) [Mic08b] is a standard way for accessing data sources. In order to intercept SQL statements generated from ADO, we utilized a technique named *universal delegator* introduced by Brown [Bro99], which allows delegator hooks to be attached to any COM objects. A delegator hook is a COM object that acts on behalf of the delegated object, and thus gets to pre and post-process COM calls. To automatically attach hooks to an ADO object at creation time, we built a custom ADO class factory, and registered this component in place of ADO. By altering the system registry, this custom ADO class factory gets loaded and/or called when requests for a particular ADO object are made. The custom ADO class factory then loads the real ADO component, calls the real class factory, and finally attaches a delegator hook to the newly created ADO object. These interception mechanisms are also transparent to web applications.

6 Evaluation

We evaluated SQLPrevent 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 [HO05, SW06, HOM06, BBMV07, KKH$^+$07].

6.1 Experimental Setup

The experimental set up is illustrated in Figure 6. 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 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 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`" clause 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 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.

SQLIA detector module threw an exception (`java.sql.SQLException`) each time it detected an SQLIA. The tested 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.

6.2 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.

6.3 Efficiency

We measured 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 set up common to both modes and 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

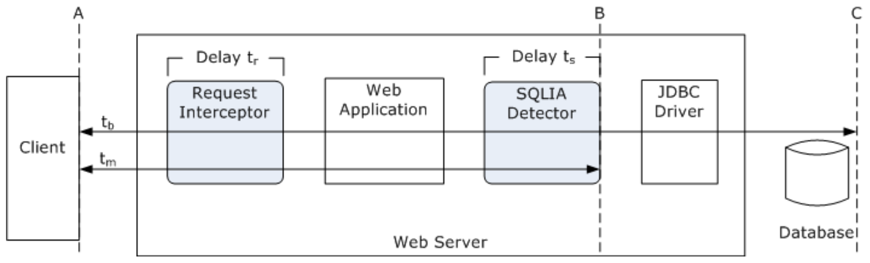


Figure 7: Detection and prevention performance evaluation. t_b and t_m are round-trip response time with SQLPrevent deployed, measured using benign and malicious requests, respectively.

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, and round-trip latency was less than 0.1 millisecond on average

6.3.1 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 7, and applied the following formula: $Detection\ Overhead = (t_r + t_s)/t_b$, where t_r and t_s are the time delays for request interceptor and SQLIA detector, respectively, and t_b is round-trip (between A to C in Figure 7) response time when a benign SQL statement is detected.

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.

Based on the experimental results, the average performance overhead for the high-end configuration was about five times higher percentage-wise than in the case

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

of low-end environment. This was likely because the response time for the high-end configuration was about 15 times smaller than for the low-end one. Therefore, we report the performance overhead for the high-end configuration only.

For each web application, Table 3 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.

6.3.2 Concurrent Access

To test SQLPrevent performance overhead under a high volume of simultaneous accesses, we used JMeter [Apa07], 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 on 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.

6.3.3 SQLPrevent vs. Other Approaches

Due to the differences in physical settings, we cannot compare SQLPrevent performance directly with those approaches that were also evaluated by their authors using AMNESIA testbed. Therefore, we list the performance data of the latter here for reference purposes only. Halfond and Orso [HO05] state that they "*found that the overhead imposed by [AMNESIA] . . . is negligible and, in fact, barely measurable, ranging from 10 to 40 milliseconds*" without providing detailed information regarding the physical settings and how the 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 average overhead for each application ranged from 2.478ms to 3.368ms. SQLCheck do not explain how the performance 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. The performance overhead ranged from 3.2% to 40.0%.

7 Discussion

In our evaluation, SQLPrevent produced no false positives or false negatives, imposed low runtime overhead on the testbed applications, and was shown to be portable among two different databases. In addition to high detection accuracy and low performance overhead, the advantages of our technique are its automatic adoptability to developer's intentions, and its ease of integration with existing web applications.

Existing dynamic SQLIA techniques are effective; they trigger an SQLIA alarm when one keyword or operator token is detected as tainted in the intercepted SQL statement. Nevertheless, if the detected keyword token is embedded in the HTTP request by a web developer on purpose, it would be an instance of false positive. In addition to the result set sorting example in Example 1, keywords and operators such as "AND" and "OR" are frequently seen in dropdown lists or other forms of HTML elements. When presented, they are commonly used to construct dynamic SQL statements. This type of *taint-keyword* scenario can be found in organizations where security development processes are not in place due to resource restrictions, or the processes are not enforced because of the schedule pressure or organizational culture. A solution that can automatically adopt to developer's intention is therefore vital to such development environments.

Our approach (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 require modification of the runtime environment, such as .NET CLR or JVM, and (4) is independent of the back-end database used.

SQLPrevent for J2EE can be easily integrated with existing J2EE web applications by (1) deploying SQLPrevent Java library into J2EE application servers, (2) configuring `HTTP Request Interceptor` filter entry in the `web.xml`, (3) replacing the class name of the real JDBC driver with the class name of `SQL Interceptor` in the configuration settings, and (4) configuring the JVM to prepend `Taint Tracker` library in front of bootstrap class path. For ASP.NET and ASP, SQLPrevent deployment process is a matter of copying and registering the binary components.

We ported SQLIntention to ASP.NET and ASP to demonstrate the generalizabilty 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 resources might have been shifted 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 can not be modified. To support ASP, we utilizes the *universal delegator* introduced by Brown [Bro99] to intercept SQL statements generated from ADO [Mic08b]. The interception mechanism is transparent to web applications and does not require modification to the ASP runtime environment. ASP web applications have been the target of waves of massive SQLIAs since October 2007 [Lan08, Kei08, Lem08]. As a consequence of these attacks, more than half a million web pages were infected with malicious JavaScript code that redirects the visitors of compromised web sites to download malware from malicious hosts [PMRM08]. Our approach can be integrated into an existing web application with a few configuration setting changes. Security protection without additional efforts from developers and administrators is vital to the protection of legacy web applications.

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.

8 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 effort needed from web developers—such as application source code analysis/modification or modification of the runtime environment—leads to limited adoption of these countermeasures in real world. In this paper, we have presented a new approach to the design of protection mechanisms for existing web applications. The approach enables treatment of web applications as black boxes for the purpose of runtime detection and prevention of SQLIAs. This work also contributes with a design, implementation, and evaluation of the proposed approach in the forms of SQLPrevent. Our experience and evaluation of SQLPrevent indicate 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.

The approach proposed in this paper is not a replacement for all other approaches against SQLIAs. It offers an alternative point in the trade-off space, as the discussion of the related work explains. 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 analysis on the subject of SQLIA vulnerability identification and their subsequent elimination, or even the use of parameterized query APIs would be more appropriate. What our approach offers is the ability to protect existing applications effectively, efficiently, and without having to depend on the application vendors or developers. SQLPrevent could be considered somewhat more like a software-based security appliance that can be "dropped" in an application server at any time with negligible needs for its administration and operation. This "drop-and-use" property of SQLPrevent is vital to the protection of web applications where source code, qualified developers, or security development processes might not be available or practical.

For future work, we plan to apply dynamic intention discovery to prevent other types of web application attacks, and 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

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2007.

[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.

[Apa07] Apache Software Foundation. Apache JMeter. http://jakarta.apache.org/jmeter/, 2007.

[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 *Proceedings of the 14th 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 *Proceedings of the Second International Conference on Applied Cryptography and Network Security (ACNS)*, pages 292–302, June 2004.

[Bre07] Breach Security Inc. ModSecurity. http://www.modsecurity.org/, 2007.

[Bro99] Keith Brown. Building a lightweight COM interception framework part 1: The universal delegator. *Microsoft Systems Journal*, January 1999.

[BWS05] Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. SQL-Guard: Using parse tree validation to prevent SQL injection attacks. In

Proceedings of the 5th 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.

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

[HGM04] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. *Wirel. Netw.*, 10(6):643–652, 2004.

[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 *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183, Long Beach, California, USA, 2005.

[HOM06] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 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*, pages 175–185, Portland, Oregon, USA, 2006.

[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.

[JD02] Klaus Julisch and Marc Darcier. Mining intrusion detection alarms for actionable knowledge. In *Proceedings of the 8th ACM International Conference on Knowledge Discovery and Data Mining*, pages 366–375, 2002.

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[Kei08] Gregg Keizer. Huge web hack attack infects 500,000 pages. *Computerworld*, April 2008.

[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 *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, December 2007.

[Lan08] Mary Landesman. Massive SQL injection attack. *Technical report, ScanSafe Threat Alerts*, May 2008.

[Lem08] Sumner Lemon. Mass SQL injection attack hits Chinese web sites. *Computerworld*, May 2008.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 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.

[Mic07] Microsoft. Transact-SQL reference. http://msdn2.microsoft.com/en-us/library/ms189826.aspx, 2007.

[Mic08a] Microsoft. HTTPModule. http://msdn.microsoft.com/en-us/library/zec9k340(VS.71).aspx, 2008.

[Mic08b] Microsoft Corporation. ADO: ActiveX Data Objects. http://msdn.microsoft.com/en-us/library/ms805098.aspx, 2008.

[Mic08c] Microsoft Corporation. COM: Component Object Model technologies. http://www.microsoft.com/com/default.mspx, 2008.

[Mik03] Aleksandr Mikunov. Rewrite MSIL code on the fly with the .NET framework profiling API. *Microsoft MSDN Magazine*, September 2003.

[MIT08] MITRE. Common vulnerabilities and exposures list. http://cve.mitre.org/, 2008.

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

[MyS07] MySQL AB Corp. MySQL 6.0 reference manual. http://dev.mysql.com/doc/refman/6.0/en/index.html, 2007.

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

[Ora07] Oracle Corp. Oracle database PL/SQL. http://www.oracle.com/technology/tech/pl_sql/index.html, 2007.

[OWA08] OWASP. Open web application security project (OWASP) top ten project. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2008.

[PB05] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, pages 124–145, 2005.

[Pie01] Matt Pietrek. The .NET profiling API and the DNProfiler tool. *Microsoft MSDN Magazine*, December 2001.

[PMRM08] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iFRAMEs point to us. In *Proceedings of 17th USENIX Security Symposium*, Boston, Massachusetts, USA, June 22-27 2008. USENIX.

[Sek09] R. Sekar. 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, February 8–11 2009.

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

[SW06] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd Annual ACM SIGPLAN - SIGACT 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 *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005)*, pages 123–140, 2005.

[Wal07] Larry Wall. perlsec - perl security. Library v.5.10, perl.org, 2007.

[WHM$^+$08] Rodrigo Werlinger, Kirstie Hawkey, Kasia Muldner, Pooya Jaferian, and Konstantin Beznosov. 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)*, pages 107–116, Pittsburgh, PA, July 23-25 2008.

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

[Zet09] Kim Zetter. 'The Analyzer' hack probe widens; $10 million allegedly stolen from U.S. banks. *Wired Magazine*, March 24 2009.