SQL Injection Attacks and Its Counter Measures

Er. Upinder Kaur [1], Er. Navdeep Kochhar[2]

Dept. of Computer Science, Baba Farid college, Deon , Bathinda, Punjab, India.
raj_chawla94@yahoo.com[1], er_navdeepkochhar@rediffmail.com[2]

Abstract: SQL injection is a technique for exploiting web applications that use client-supplied data in SQL queries, but without first stripping potentially harmful characters. Despite being remarkably simple to protect against, there is an astonishing number of production systems connected to the Internet that are vulnerable to this type of attack. To address this problem, we present an extensive review of the different types of SQL injection attacks known to date. For each type of attack, we provide descriptions and examples of how attacks of that type could be performed. We also present and analyze existing detection and prevention techniques against SQL injection attacks. For each technique, we discuss its strengths and weaknesses in addressing the entire range of SQL injection attacks

Keywords: SQLIAS, preventions, attacks, SQL injections.

1. Introduction:

SQL injection vulnerabilities have been described as one of the most serious threats for Web applications [3][11]. Web applications that are vulnerable to SQL injection may allow an attacker to gain complete access to their underlying databases. Because these databases often contain sensitive consumer or user information, the resulting security violations can include identity theft, loss of confidential information, and fraud. In some cases, attackers can even use an SQL injection vulnerability to take control of and corrupt the system that hosts the Web application. Web applications that are vulnerable to SQL Injection Attacks (SQLIAs) are widespread—a study by Gartner Group on over 300 Internet Web sites has shown that most of them could be vulnerable to SQLIAs. In fact, SQLIAs have successfully targeted high-profile victims such as Travelocity, FTD.com, and Guess Inc. SQL injection refers to a class of code-injection attacks in which data provided by the user is included in an SQL query in such a way that part of the user's input is treated as SQL code. By leveraging these vulnerabilities, an attacker can submit SQL commands directly to the database. These attacks are a serious threat to any Web application that receives input from users and incorporates it into SQL queries to an underlying database. Most web applications used on the Internet or within enterprise systems work this way and could therefore be vulnerable to SQL injection. The cause of SQL injection vulnerabilities is relatively simple and well understood: insufficient validation of user input. To address this problem, developers have proposed a range of coding guidelines (e.g., [18]) that promote defensive coding practices, such as encoding user input and validation. A rigorous and systematic application of these techniques is an effective solution for preventing SQL injection vulnerabilities. However, in practice, the application of such techniques is human-based and, thus, prone to errors. Furthermore, fixing legacy code-bases that might contain SQL injection vulnerabilities can be an extremely labor-intensive task. Although recently there has been a great deal of attention to the problem of SQL injection vulnerabilities, many proposed solutions fail to address the full scope of the problem. There are many types of SQLIAs and countless variations on these basic types. Researchers and practitioners are often unaware of the myriad of different techniques that can be used to perform SQLIAs. Therefore, most of the solutions proposed detect or prevent only a subset of the possible SQLIAs. To address this problem, we present a comprehensive survey of SQL injection attacks known to date. To compile the survey, we used information gathered from various sources, such as papers,Web sites, mailing lists, and experts in the area. For each attack type considered, we give a characterization of the attack, illustrate its effect, and provide examples of how that type of attack could be performed. This set of attack types is then used to evaluate state of the art detection and prevention techniques and compare their strengths and weaknesses. The results of this comparison show the effectiveness of these techniques.

1.1 Injection Mechanisms

Malicious SQL statements can be introduced into a vulnerable application using many different input mechanisms. In this section, we explain the most common mechanisms.

Injection through user input: In this case, attackers inject SQL commands by providing suitably crafted user input. A Web application can read user input in several ways based on the environment in which the application is deployed. In most SQLIAs that target Web applications, user input typically comes from form submissions that are sent to the Web application via HTTP GET or POST requests [14]. Web applications are generally able to access the user input contained in these requests as they would access any other variable in the environment.

Injection through cookies: Cookies are files that contain state information generated byWeb applications and stored on the client machine. When a client returns to a Web application, cookies can be used to restore the client’s state information. Since the client has control over the storage of the cookie, a malicious client could tamper with the cookie’s contents. If a Web application uses the cookie’s contents to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie [8].

Injection through server variables: Server variables are a collection of variables that contain HTTP, network headers, and environmental variables. Web applications use these server variables in a variety of ways, such as logging usage statistics and identifying browsing trends. If these variables are logged to a database without sanitization, this could create an SQL injection vulnerability. Because attackers can forge the values that are placed in HTTP and network headers, they can exploit this vulnerability by placing an SQLIA directly into the headers. When the query to log the server variable is issued to the database, the attack in the forged header is then triggered.

Second-order injection: In second-order injections, attackers seed malicious inputs into a system or database to indirectly trigger an SQLIA when that input is used at a later time. The objective of this kind of attack differs significantly from a regular (i.e., first order) injection attack. Second-order injections are not trying to cause the attack to occur when the malicious input initially reaches the database. Instead, attackers rely on knowledge of where the input will be subsequently used and craft their attack so that it occurs during that usage. To clarify, we present a classic example of a second order injection attack (taken from [1]). In the example, a user registers on a website using a seeded user name, such as “admin’ -- ”. The application properly escapes the single quote in the input before storing it in the database, preventing its potentially malicious effect. At this point, the user modifies his or her password, an operation that typically involves (1) checking that the user knows the current password and (2) changing the password if the check is successful. To do this, the Web application might construct an SQL command as follows:

queryString="UPDATE users SET password=’" + newPassword +
"’ WHERE userName=’" + userName + "’ AND password=’" +
oldPassword + "’" newPassword and oldPassword are the new and old passwords,
respectively, and userName is the name of the user currently
logged-in (i.e., ‘‘admin’--’’).
Therefore, the query string that is sent to the database is (assume that newPassword and oldPas-sword are “newpwd” and“oldpwd”):
UPDATE users SET password=’newpwd’ WHERE userName= ’admin’--’ AND password=’oldpwd’

Because “--” is the SQL comment operator, everything after it is ignored by the database. Therefore, the result of this query is that the database changes the password of the administrator (“admin”) to an attacker-specified value. Second-order injections can be especially difficult to detect and prevent because the point of injection is different from the point where the attack actually manifests itself. A developer may properly escape, type-check, and filter input that comes from the user and assume it is safe. Later on, when that data is used in a different context, or to build a different type of query, the previously sanitized input may result in an injection attack.

2. Different types of SQLIA

In this section, we present and discuss the different kinds of SQLIAs known to date. For each attack type, we provide a descriptive name, a description of the attack, an attack example, and a set of references to publications and Web sites that discuss the attack technique and its variations in greater detail. The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific goals of the attacker. Note also that there are countless variations of each attack type. For space reasons, we do not present all of the possible attack variations but instead present a single representative example.

Tautologies Attack: Bypassing authentication, identifying injectable parameters, extracting data.
Description: The general goal of a tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an injectable field that is used in a query's WHERE conditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the query to be returned. Example: In this example attack, an attacker submits " ' or 1=1 - - " for the login input field (the input submitted for the other fields is irrelevant). The resulting query is:

SELECT accounts FROM users WHERE login='' or 1=1 -- AND pass='' AND pin=

The code injected in the conditional (OR 1=1) transforms the entire WHERE clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them. In our example, the returned set evaluates to a non null value, which causes the application to conclude that the user authentication was successful. Therefore, the application would invoke method displayAccounts() and show all of the accounts in the set returned by the database. [1][28][21][18]

Illegal/Logically Incorrect Queries Attack: Identifying injectable parameters, performing database finger-printing, extracting data.
Description: This attack lets an attacker gather important information about the type and structure of the back-end database of a Web application. The attack is considered a preliminary, information gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, the simple fact that an error messages is generated can often reveal vulnerable/injectable parameters to an attacker. Additional error information, originally intended to help programmers debug their applications, further helps attackers gain information about the schema of the back-end database.
Example: This example attack's goal is to cause a type conversion error that can reveal relevant data. To do this, the attacker injects the following text into input field pin: "convert(int,(select top 1 name from sysobjects where xtype='u'))". The resulting query is:

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

In the attack string, the injected select query attempts to extract the first user table (xtype='u') from the database's metadata table (assume the application is using Microsoft SQL Server, for which the metadata table is called sysobjects). The query then tries to convert this table name into an integer. Because this is not a legal type conversion, the database throws an error. For Microsoft SQL Server, the error would be: "Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int." There are two useful pieces of information in this message that aid an attacker. First, the attacker can see that the database is an SQL Server database, as the errormessage explicitly states this fact. Second, the error message reveals the value of the string that caused the type conversion to occur. In this case, this value is also the name of the first user-defined table in the database: "CreditCards." A similar strategy can be used to systematically extract the name and type of each column in the database. Using this information about the schema of the database, an attacker can then create further attacks that target specific pieces of information. [1][22][28]

Union Query Attack: Bypassing Authentication, extracting data.
Description: In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. Attackers do this by injecting a statement of the form: UNION SELECT <rest of injected query>. Because the attackers completely control the second/injected query, they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.
Example: Referring to the running example, an attacker could inject the text "' UNION SELECT cardNo from CreditCards where acctNo=10032 - -" into the login field, which produces the following query:

SELECT accounts FROM users WHERE login='' UNION SELECT cardNo from CreditCards where acctNo=10032 -- AND pass='' AND pin=

Assuming that there is no login equal to “”, the original first query returns the null set, whereas the second query returns data from the “CreditCards” table. In this case, the database would return column “cardNo” for account “10032.” The database takes the results of these two queries, unions them, and returns them to the application. In many applications, the effect of this operation is that the value for “cardNo” is displayed along with the account information. [1][28][21]

PiggyBacked Queries Attack: Extracting data, adding or modifying data, performing denial of service, executing remote commands.
Description: In this attack type, an attacker tries to inject additional queries into the original query. We distinguish this type from others because, in this case, attackers are not trying to modify the original query; instead, they are trying to include new and distinct queries that “piggy-back” on the original query. As a result, the database receives multiple SQL queries. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string.
Example: If the attacker inputs “’; drop table users - - ” into the pass field, the application generates the query:

```
SELECT accounts FROM users WHERE login=’doe’
AND pass=’’; drop table users -- ’ AND pin=123
```

After completing the first query, the database would recognize the 1 stored procedures are routines stored in the database and run by the database engine. These procedures can be either user-defined procedures or procedures provided by the database by default. query delimiter (“;”) and execute the injected second query. The result of executing the second query would be to drop table users, which would likely destroy valuable information. Other types of queries could insert new users into the database or execute stored procedures. Note that many databases do not require a special character to separate distinct queries, so simply scanning for a query separator is not an effective way to prevent this type of attack. [1][28][18]

Stored Procedures Attack: Performing privilege escalation, performing denial of service, executing remote commands.
Description: SQLIAs of this type try to execute stored procedures present in the database. Today, most database vendors ship databases with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Therefore, once an attacker determines which backend database is in use, SQLIAs can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system. It is a common misconception that using stored procedures to write Web applications renders them invulnerable to SQLIAs. Developers are often surprised to find that their stored procedures can be just as vulnerable to attacks as their normal applications [18][24]. Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run arbitrary code on the server or escalate their privileges [9].

Stored procedure for checking credentials.

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

Example: This example demonstrates how a parameterized stored procedure can be exploited via an SQLIA. In the example, we assume that the query string constructed at lines of our example has been replaced by a call to the stored procedure. The stored procedure returns a true/false value to indicate whether the user’s credentials authenticated correctly. To launch an SQLIA, the attacker simply injects “ ’ ; SHUTDOWN; - -” into either the userName or password fields. This injection causes the stored procedure to generate the following query:

```
SELECT accounts FROM users WHERE login=’doe’
AND pass=’ ’; SHUTDOWN; -- AND pin=
```

At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second, malicious query is executed, which results in a database shut down. This example shows that stored procedures can be vulnerable to the same range of attacks as traditional application code. [1][4][9][10][24][28][21][18]

Inference Attack: dentifying injectable parameters, extracting data, determining database schema.
Description: In this attack, the query is modified to recast it in the form of an action that is executed based on the answer to a true/- false question about data values in the database. In this type of injection, attackers are generally trying to attack a site that has been secured enough so that, when an injection has

succeeded, there is no usable feedback via database error messages. In this situation, the attacker injects commands into the site and then observes how the function/response of the website changes. By carefully noting when the site behaves the same and when its behavior changes, the attacker can deduce not only whether certain parameters are vulnerable, but also additional information about the values in the database. There are two well known attack techniques that are based on inference. They allow an attacker to extract data from a database and detect vulnerable parameters. Researchers have reported that with these techniques they have been able to achieve a data extraction rate of 1B/s [2].

Blind Injection: In this technique, the information must be inferred from the behavior of the page by asking the server true/- false questions. If the injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally-functioning page.

Timing Attacks: A timing attack allows an attacker to gain information from a database by observing timing delays in the response of the database. This attack is very similar to blind injection, but uses a different method of inference. To perform a timing attack, attackers structure their injected query in the form of an if/then statement, whose branch predicate corresponds to an unknown about the contents of the database. Along one of the branches, the attacker uses a SQL construct that takes a known amount of time to execute, (e.g. the WAITFOR keyword, which causes the database to delay its response by a specified time). By measuring the increase or decrease in response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injected question.

Example: Using the code from our running example, we illustrate two ways in which Inference based attacks can be used. The first of these is identifying injectable parameters using blind injection. Consider two possible injections into the login field. The first being “legalUser’ and 1=0 - -” and the second, “legalUser’ and 1=1 - -”. These injections result in the following two queries:

SELECT accounts FROM users WHERE login=’legalUser’ and 1=0 -- ’ AND pass=’’ AND pin=0

SELECT accounts FROM users WHERE login=’legalUser’ and 1=1 -- ’ AND pass=’’ AND pin=0

Now, let us consider two scenarios. In the first scenario, we have a secure application, and the input for login is validated correctly. In this case, both injections would return login error messages, and the attacker would know that the login parameter is not vulnerable. In the second scenario, we have an insecure application and the login parameter is vulnerable to injection. The attacker submits the first injection and, because it always evaluates to false, the application returns a login error message. At this point however, the attacker does not know if this is because the application validated the input correctly and blocked the attack attempt or because the attack itself caused the login error. The attacker then submits the second query, which always evaluates to true. If in this case there is no login error message, then the attacker knows that the attack went through and that the login parameter is vulnerable to injection.

The second way inference based attacks can be used is to perform data extraction. Here we illustrate how to use a Timing based inference attack to extract a table name from the database. In this attack, the following is injected into the login parameter: ‘‘legalUser’ and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 --’’. This produces the following query:

SELECT accounts FROM users WHERE
login=’legalUser’ and
ASCII(SUBSTRING((select top 1 name from
sysobjects),1,1))
> X WAITFOR 5 -- ’ AND pass=’’ AND pin=0

In this attack the SUBSTRING function is used to extract the first character of the first table’s name. Using a binary search strategy, the attacker can then ask a series of questions about this character. In this case, the attacker is asking if the ASCII value of the character is greater-than or less-than-or-equal-to the value of X. If the value is greater, the attacker knows this by observing an additional 5 second delay in the response of the database. The attacker can then use a binary search by varying the value of X to identify the value of the first character. [2]

Alternate Encodings Attack : Evading detection.
Description: In this attack, the injected text is modified so as to avoid detection by defensive coding practices and also many automated prevention techniques. This attack type is used in conjunction

with other attacks. In other words, alternate encodings do not provide any unique way to attack an application; they are simply an enabling technique that allows attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known “bad characters,” such as single quotes and comment operators. To evade this defense, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Common scanning and detection techniques do not try to evaluate all specially encoded strings, thus allowing these attacks to go undetected. The application may scan for certain types of escape characters that represent alternate encodings in its language domain. Another layer (e.g., the database) may use different escape characters or even completely different ways of encoding. For example, a database could use the expression char(120) to represent an alternately-encoded character “x”, but char(120) has no special meaning in the application language’s context. An effective code-based defense against alternate encodings is difficult to implement in practice because it requires developers to consider of all of the possible encodings that could affect a given query string as it passes through the different application layers. Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings.

Example: Because every type of attack could be represented using an alternate encoding, here we simply provide an example (see [18]) of how esoteric an alternatively-encoded attack could appear. In this attack, the following text is injected into the login field: “legalUser’; exec(0x73687574646f776e) - - ”. The resulting query generated by the application is:

```
SELECT accounts FROM users WHERE
login=’legalUser’;nexec(char(0x73687574646f776e))
-- AND pass=’’ AND pin=
```

This example makes use of the char() function and of ASCII hexadecimal encoding. The char() function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the string “SHUTDOWN.” Therefore, when the query is interpreted by the database, it would result in the execution, by the database, of the SHUTDOWN command. [1][18]

3. PREVENTION OF SQLIAS

Researchers have proposed a wide range of techniques to address the problem of SQL injection. These techniques range from development best practices to fully automated frameworks for detecting and preventing SQLIAs. In this section, we review these proposed techniques and summarize the advantages and disadvantages associated with each technique.

3.1 Defensive Coding Practices

The root cause of SQL injection vulnerabilities is insufficient input validation. Therefore, the straightforward solution for eliminating these vulnerabilities is to apply suitable defensive coding practices. Here, we summarize some of the best practices proposed in the literature for preventing SQL injection vulnerabilities.

Input type checking: SQLIAs can be performed by injecting commands into either a string or numeric parameter. Even a simple check of such inputs can prevent many attacks. For example, in the case of numeric inputs, the developer can simply reject any input that contains characters other than digits. Many developers omit this kind of check by accident because user input is almost always represented in the form of a string, regardless of its content or intended use.

Encoding of inputs: Injection into a string parameter is often accomplished through the use of meta-characters that trick the SQL parser into interpreting user input as SQL tokens. While it is possible to prohibit any usage of these meta-characters, doing so would restrict a non-malicious user’s ability to specify legal inputs that contain such characters. A better solution is to use functions that encode a string in such a way that all meta-characters are specially encoded and interpreted by the database as normal characters.

Positive pattern matching: Developers should establish input validation routines that identify good input as opposed to bad input. This approach is generally called positive validation, as opposed to negative validation, which searches input for forbidden patterns or SQL tokens. Because developers might not be able to envision every type of attack that could be launched against their application, but should be able to specify all the forms of legal input, positive validation is a safer way to check inputs.

Identification of all input sources: Developers must check all input to their application. There are many possible sources of input to an application. If used to construct a query, these input sources can be a way for an attacker to introduce an SQLIA. Simply put, all input sources must be checked. Although defensive coding practices remain the best way to prevent SQL injection vulnerabilities, their application is problematic in practice. Defensive coding is prone to human error and is not as rigorously and completely applied as automated techniques. While most developers do make an effort to code safely, it is extremely difficult to apply defensive coding practices rigorously and correctly to all sources of input. In fact, many of the SQL injection vulnerabilities discovered in real applications are due to human errors: developers forgot to add checks or did not perform adequate input validation [20][23][33]. In other words, in these applications, developers were making an effort to detect and prevent SQLIAs, but failed to do so adequately and in every needed location. These examples provide further evidence of the problems associated with depending on developer's use of defensive coding. Moreover, approaches based on defensive coding are weakened by the widespread promotion and acceptance of so-called "pseudoremedies" [18]. We discuss two of the most commonly-proposed pseudo-remedies. The first of such remedies consists of checking user input for SQL keywords, such as "FROM," "WHERE," and "SELECT," and SQL operators, such as the single quote or comment operator. The rationale behind this suggestion is that the presence of such keywords and operators may indicate an attempted SQLIA. This approach clearly results in a high rate of false positives because, in many applications, SQL keywords can be part of a normal text entry, and SQL operators can be used to express formulas or even names (e.g., O'Brian). The second commonly suggested pseudo-remedy is to use stored procedures or prepared statements to prevent SQLIAs. Unfortunately, stored procedures and prepared statements can also be vulnerable to SQLIAs unless developers rigorously apply defensive coding guidelines. Interested readers may refer to [1][25][28][29] for examples of how these pseudo-remedies can be subverted.

3.2 Detection and Prevention Techniques

Researchers have proposed a range of techniques to assist developers and compensate for the shortcomings in the application of defensive coding.

Black Box Testing: Huang and colleagues [19] proposeWAVES, a black-box technique for testing Web applications for SQL injection vulnerabilities. The technique uses a Web crawler to identify all points in a Web application that can be used to inject SQLIAs. It then builds attacks that target such points based on a specified list of patterns and attack techniques. WAVES then monitors the application's response to the attacks and uses machine learning techniques to improve its attack methodology. This technique improves over most penetration-testing techniques by using machine learning approaches to guide its testing. However, like all black-box and penetration testing techniques, it cannot provide guarantees of completeness.

Static Code Checkers: JDBC-Checker is a technique for statically checking the type correctness of dynamically-generated SQL queries [12][13]. This technique was not developed with the intent of detecting and preventing general SQLIAs, but can nevertheless be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. JDBC-Checker is able to detect one of the root causes of SQLIA vulnerabilities in code— improper type checking of input. However, this technique would not catch more general forms of SQLIAs because most of these attacks consist of syntactically and type correct queries. Wassermann and Su propose an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

Combined Static and Dynamic Analysis: AMNESIA is a model-based technique that combines static analysis and runtime monitoring [17][16]. In its static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the database. In its dynamic phase, AMNESIA intercepts all queries before they are sent to the database and checks each query against the statically built models. Queries that violate the model are identified as SQLIAs and prevented from executing on the database. In their evaluation, the authors have shown that this technique performs well against SQLIAs. The primary limitation of this technique is that its success is dependent on the accuracy of its static analysis for building query models. Certain types of code obfuscation or query development techniques could make this step less precise and result in both false positives and false negatives. Similarly, two recent related approaches, SQLGuard [6] and SQLCheck also check queries at runtime to see if they conform to a model of expected queries. In these approaches, the model is expressed as a grammar that only accepts legal queries. In SQLGuard,the model is deduced at runtime by examining the structure of the query before and after the addition of user-input. In SQLCheck, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. Additionally, the use of these two approaches requires the developer to either rewrite code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

Taint Based Approaches: WebSSARI detects input-validation related errors using information flow analysis [20]. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application to satisfy these preconditions. The WebSSARI system works by considering as sanitized input that has passed through a predefined set of filters. In their evaluation, the authors were able to detect security vulnerabilities in a range of existing applications. The primary drawbacks of this technique are that it assumes that adequate preconditions for sensitive functions can be accurately expressed using their typing system and that having input passing through certain types of filters is sufficient to consider it not tainted. Formany types of functions and applications, this assumption is too strong. Livshits and Lam [23] use static analysis techniques to detect vulnerabilities in software. The basic approach is to use information flow techniques to detect when tainted input has been used to construct an SQL query. These queries are then flagged as SQLIA vulnerabilities. The authors demonstrate the viability of their technique by using this approach to find security vulnerabilities in a benchmark suite. The primary limitation of this approach is that it can detect only known patterns of SQLIAs and, because it uses a conservative analysis and has limited support for untainting operations, can generate a relatively high amount of false positives. Several dynamic taint analysis approaches have been proposed. Two similar approaches by Nguyen-Tuong and colleagues and Pietraszek and Berghe modify a PHP interpreter to track precise per-character taint information. The techniques use a context sensitive analysis to detect and reject queries if untrusted input has been used to create certain types of SQL tokens. A common drawback of these two approaches is that they require modifications to the runtime environment, which affects portability. A technique by Haldar and colleagues [15] and SecuriFly [26] implement a similar approach for Java. However, these techniques do not use the context sensitive analysis employed by the other two approaches and track taint information on a per-string basis (as opposed to percharacter). SecuriFly also attempts to sanitize query strings that have been generated using tainted input. However, this sanitization approach does not help if injection is performed into numeric fields. In general, dynamic taint-based techniques have shown a lot of promise in their ability to detect and prevent SQLIAs. The primary drawback of these approaches is that identifying all sources of tainted user input in highly-modular Web applications and accurately propagating taint information is often a difficult task.

NewQueryDevelopmentParadigms: Two recent approaches, SQL DOM [27] and Safe Query Objects [7], use encapsulation of database queries to provide a safe and reliable way to access databases. These techniques offer an effective way to avoid the SQLIA problem by changing the query-building process from an unregulated one that uses string concatenation to a systematic one that uses a type-checked API.Within their API, they are able to systematically apply coding best practices such as input filtering and rigorous type checking of user input. By changing the development paradigm in which SQL queries are created, these techniques eliminate the coding practices that make most SQLIAs possible. Although effective, these techniques have the drawback that they require developers to learn and use a new programming paradigm or query-development process. Furthermore, because they focus on using a new development process, they do not provide any type of protection or improved security for existing legacy systems.

Intrusion Detection Systems: Valeur and colleagues [29] propose the use of an Intrusion Detection System(IDS) to detect SQLIAs. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model. In their evaluation, Valeur and colleagues have shown that their system is able to detect attacks with a high rate of success. However, the fundamental limitation of learning based techniques is that they can provide no guarantees about their detection abilities because their success is dependent on the quality of the training set used. A poor training set would cause the learning technique to generate a large number of false positives and negatives.

Proxy Filters: Security Gateway [28] is a proxy filtering system that enforces input validation rules on the data flowing to a Web application. Using their Security Policy Descriptor Language (SPDL), developers provide constraints and specify transformations to be applied to application parameters as they flow from the Web page to the application server. Because SPDL is highly expressive, it allows developers considerable freedom in expressing their policies. However, this approach is human-based and, like defensive programming, requires developers to know not only which data needs to be filtered, but also what patterns and filters to apply to
the data.

Instruction Set Randomization: SQLrand [5] is an approach based on instruction-set randomization. SQLrand provides a framework that allows developers to create queries using randomized instructions instead of normal SQL keywords. A proxy filter intercepts queries to the database and de-randomizes the keywords. SQL code injected by an attacker would not have been constructed using the randomized instruction set. Therefore, injected commands would result in a syntactically incorrect query. While this technique can be very effective, it has several practical drawbacks. First, since it uses a secret key to modify instructions, security of the approach is dependent on attackers not being able to discover the key. Second, the approach imposes a significant infrastructure overhead because it require the integration of a proxy for the database in the system.

4. TECHNIQUES EVALUATION

In this section, we evaluate the techniques presented in Section 3 using several different criteria. We first consider which attack types each technique is able to address. For the subset of techniques that are based on code improvement, we look at which defensive coding practices the technique helps enforce. We then identify which injection mechanism each technique is able to handle. Finally, we evaluate the deployment requirements of each technique.

4.1 Evaluation with Respect to Attack Types

We evaluated each proposed technique to assess whether it was capable of addressing the different attack types presented in Section 2. For most of the considered techniques, we did not have access to an implementation because either the technique was not implemented or its implementation was not available. Therefore, we evaluated the techniques analytically, as opposed to evaluating them against actual attacks. For developer-based techniques, that is, those that required developer intervention, we assumed that the developers were able to correctly apply all required defensive coding practices. In other words, our assessment of these techniques is optimistic compared to what their performance may be in practice. In our tables, we denote developer-based techniques with the symbol "*". For the purposes of the comparison, we divide the techniques into two groups: prevention-focused and detection-focused techniques. Prevention-focused techniques are techniques that statically identify vulnerabilities in the code, propose a different development paradigm for applications that generate SQL queries, or add checks to the application to enforce defensive coding best practices. Detection-focused techniques are techniques that detect attacks mostly at runtime. Tables 1 and 2 summarize the results of our evaluation. We use four different types of markings to indicate how a technique performed with respect to a given attack type. We use the symbol "•" to denote that a technique can successfully stop all attacks of that type. Conversely, we use the symbol "×" to denote that a technique is not able to stop attacks of that type. We used two different symbols to classify techniques that are only partially effective. The symbol "◦" denotes a technique that can address the attack type considered, but cannot provide any guarantees of completeness. An example of one such technique would be a black-box testing technique such as WAVES [19] or the IDS based approach from Valeur and colleagues [29]. The symbol "−," denotes techniques that address the attack type considered only partially because of intrinsic limitations of the underlying approach. For example, JDBCChecker [12][13] detects type-related errors that enable SQL injection vulnerabilities. However, because type-related errors are only one of the many possible causes of SQL injection vulnerabilities, this approach is classified as only partially handling each attack type. Half of the prevention-focused techniques effectively handle all of the attack types considered. Some techniques are only partially effective: JDBC-Checker by definition addresses only a subset of SQLIAs; Security Gateway, because it cannot handle all of the injection sources cannot completely address all of the attack profiles; SecuriFly, because its prevention method is to escape all SQL meta-characters, which still would allow injection into numeric fields; and WAVES, which because it is a testing based technique, cannot provide guarantees as to its completeness. We believe that, overall, the prevention-focused techniques performed well because they incorporate the defensive coding practices in their prevention mechanisms. See Section 4.4 for further discussion on this topic. Most of the detection-focused techniques perform fairly uniformly against the various attack types. The three exceptions are the IDSbased approach by Valeur and colleagues [29], whose effectiveness depends on the quality of the training set used, Java Dynamic Tainting [15], whose performance is negatively affected by the fact that its untainting operations allow input to be used without regard to the quality of the check, and Tautology-checker, which by definition can only address tautology-based attacks. Two attack types, stored procedures and alternate encodings, caused problems for most techniques. With stored procedures, the code that generates the query is stored and executed on the database. Most of the techniques considered focused only on queries generated within the application. Expanding the techniques to also encompass the queries generated and executed on the database is not straightforward and would, in general, require substantial effort. For this reason, attacks based on stored procedures are problematic for many techniques. Attacks based on alternate encoding are also difficult to handle. Only three techniques, AMNESIA, SQLCheck, and SQLGuard explicitly address these types of attacks. The reason why these techniques are successful against such attacks is that they use the database lexer or parser to interpret a query string in the same way that the database would. Other techniques that score well in this category are either developer-based techniques (i.e., Java Static Tainting and WebSSARI) or techniques that address the problem by using a standard API (i.e., SQL DOM and Safe Query Objects). It is important to note that we did not take precision into account in our evaluation. Many of the techniques that we consider are based on some conservative analysis or assumptions that may result in false positives. However, because we do not have an accurate way to classify the accuracy of such techniques, short of implementing all of them and assessing their performance on a large set of legitimate inputs, we have not considered this characteristic in our assessment.

4.2 Evaluation with Respect to Injection Mechanisms

We assessed each of the techniques with respect to their handling of the various injection mechanisms that we defined in Section 1.1. Although most of the techniques do not specifically address all of those injection mechanisms, all but two of them could be easily extended to handle all such mechanisms. The two exceptions are Security Gateway and WAVES. Security Gateway can examine only URL parameters and cookie fields. Because it resides on the network between the application and the attacker, it cannot examine server variables and second-order injection sources, which do not pass through the gateway. WAVES can only address injection through user input because it only generates attacks that can be submitted to the application via the Web page forms.

4.3 Evaluation with Respect to Deployment Requirements

Each of the techniques have different deployment requirements. To determine the effort and infrastructure required to use the technique, we examined the author's description of the technique and its current implementation. We evaluated each technique with respect to the following criteria: (1) Does the technique require developers to modify their code base? (2) What is the degree of automation of the detection aspect of the approach? (3) What is the degree of automation of the prevention aspect of the approach? (4) What infrastructure (not including the tool itself) is needed to successfully use the technique? The results of this classification are summarized in Table 3.

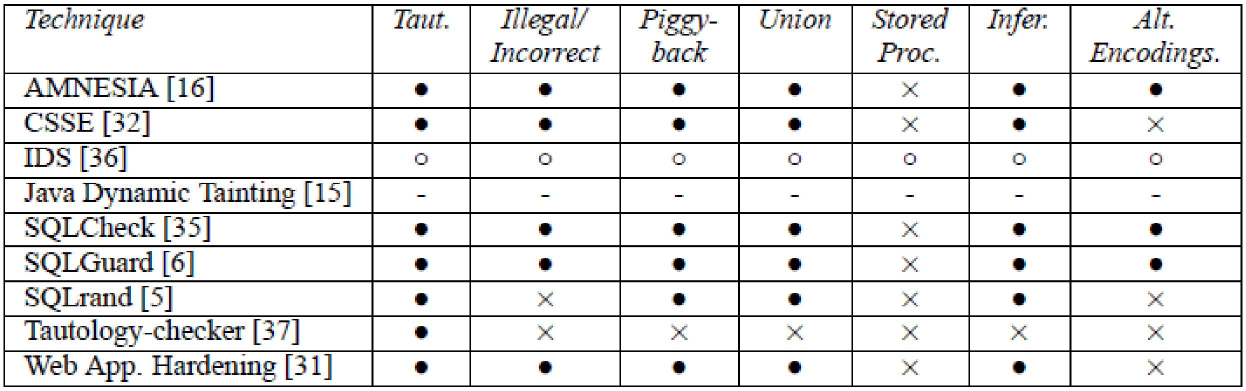
4.4 Evaluation of PreventionFocused Techniques with Respect to Defensive Coding Practices

Our initial evaluation of the techniques against the various attack types indicates that the prevention-focused techniques perform very well against most of these attacks. We hypothesize that this result is due to the fact that many of the prevention techniques are actually applying defensive coding best practices to the code base. Therefore, we examine each of the prevention-focused techniques and classify them with respect to the defensive coding practice that they enforce. Not surprisingly, we find that these techniques enforce many of these practices. Table 4 summarizes, for each technique, which of the defensive coding practices it enforces.

5. Future work and Conclusion:

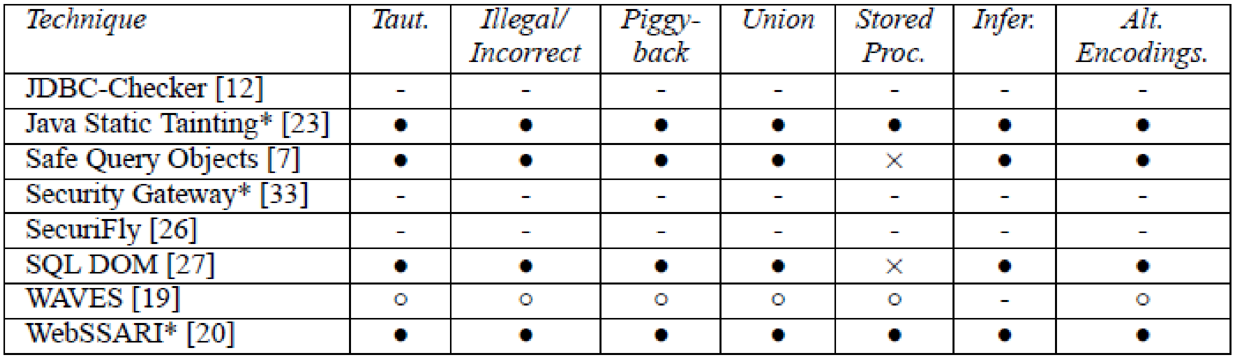
In this paper, we have presented a survey and comparison of current techniques for detecting and preventing SQLIAs. To perform this evaluation, we first identified the various types of SQLIAs known to date. We then evaluated the considered techniques in terms of their ability to detect and/or prevent such attacks. We also studied the different mechanisms through which SQLIAs can be introduced into an application and identified which techniques were able to handle which mechanisms. Lastly, we summarized the deployment requirements of each technique and evaluated to what extent its detection and prevention mechanisms could be fully automated. Our evaluation found several general trends in the results. Many of the techniques have problems handling attacks that take advantage of poorly-coded stored procedures and cannot handle attacks that disguise themselves using alternate encodings. We also found a general distinction in prevention abilities based on the difference between prevention-focused and general detection and prevention techniques. Section 4.4 suggests that this difference could be explained by the fact that prevention-focused techniques try to incorporate defensive coding best practices into their attack prevention mechanisms. Future evaluation work should focus on evaluating the techniques' precision and effectiveness in practice. Empirical evaluations such as those presented in related work (e.g., [17]) would allow for comparing the performance of the different techniques when they are subjected to real-world attacks and legitimate inputs.

Table 1. Comparison of detection-focused techniques with respect to attack types.



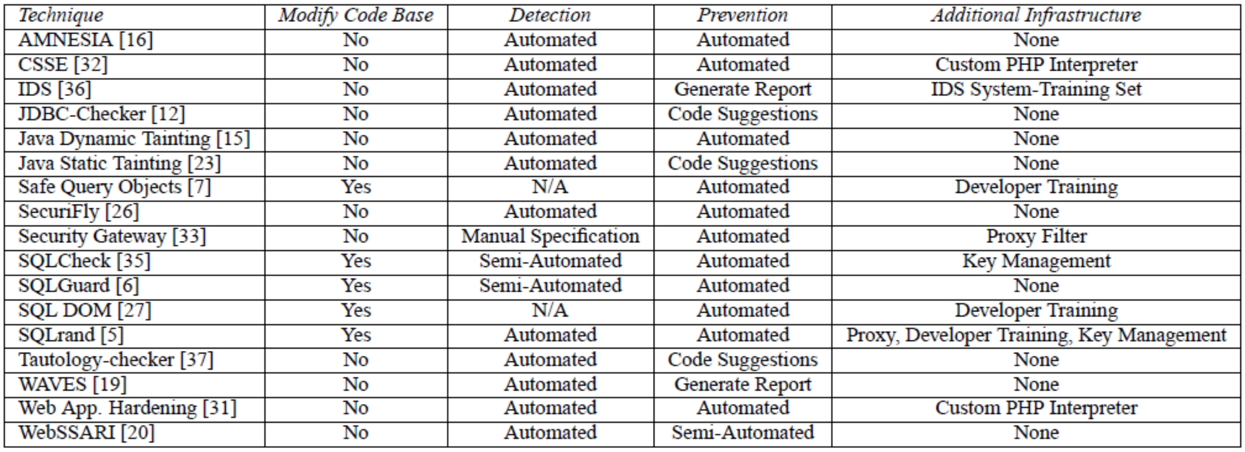
Technique	*Taut.*	*Illegal/ Incorrect*	*Piggy-back*	*Union*	*Stored Proc.*	*Infer.*	*Alt. Encodings.*
AMNESIA [16]	●	●	●	●	×	●	●
CSSE [32]	●	●	●	●	×	●	×
IDS [36]	○	○	○	○	○	○	○
Java Dynamic Tainting [15]	-	-	-	-	-	-	-
SQLCheck [35]	●	●	●	●	×	●	●
SQLGuard [6]	●	●	●	●	×	●	●
SQLrand [5]	●	×	●	●	×	●	×
Tautology-checker [37]	●	×	×	×	×	×	×
Web App. Hardening [31]	●	●	●	●	×	●	×

Table 2: Comparison of prevention-focused techniques with respect to attack types.



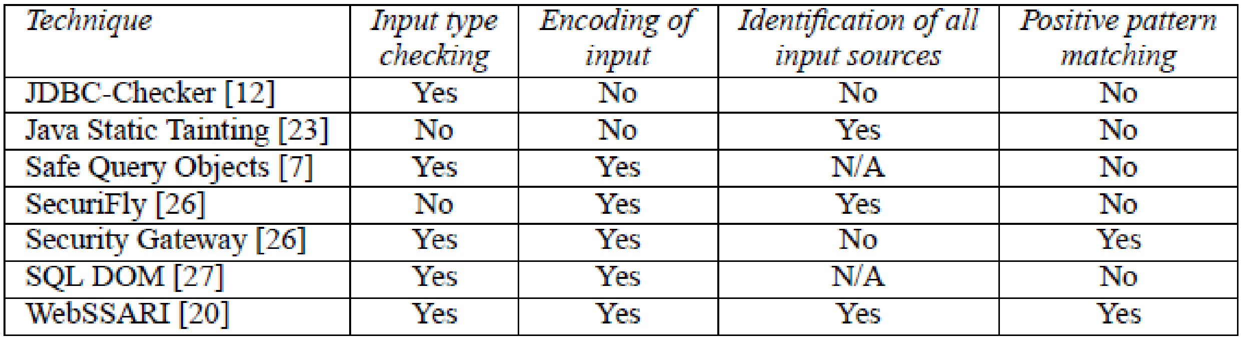
Technique	*Taut.*	*Illegal/ Incorrect*	*Piggy-back*	*Union*	*Stored Proc.*	*Infer.*	*Alt. Encodings.*
JDBC-Checker [12]	-	-	-	-	-	-	-
Java Static Tainting* [23]	●	●	●	●	●	●	●
Safe Query Objects [7]	●	●	●	●	×	●	●
Security Gateway* [33]	-	-	-	-	-	-	-
SecuriFly [26]	-	-	-	-	-	-	-
SQL DOM [27]	●	●	●	●	×	●	●
WAVES [19]	○	○	○	○	○	-	○
WebSSARI* [20]	●	●	●	●	●	●	●

Table 3: Comparison of techniques with respect to deployment requirements.



Technique	*Modify Code Base*	*Detection*	*Prevention*	*Additional Infrastructure*
AMNESIA [16]	No	Automated	Automated	None
CSSE [32]	No	Automated	Automated	Custom PHP Interpreter
IDS [36]	No	Automated	Generate Report	IDS System-Training Set
JDBC-Checker [12]	No	Automated	Code Suggestions	None
Java Dynamic Tainting [15]	No	Automated	Automated	None
Java Static Tainting [23]	No	Automated	Code Suggestions	None
Safe Query Objects [7]	Yes	N/A	Automated	Developer Training
SecuriFly [26]	No	Automated	Automated	None
Security Gateway [33]	No	Manual Specification	Automated	Proxy Filter
SQLCheck [35]	Yes	Semi-Automated	Automated	Key Management
SQLGuard [6]	Yes	Semi-Automated	Automated	None
SQL DOM [27]	Yes	N/A	Automated	Developer Training
SQLrand [5]	Yes	Automated	Automated	Proxy, Developer Training, Key Management
Tautology-checker [37]	No	Automated	Code Suggestions	None
WAVES [19]	No	Automated	Generate Report	None
Web App. Hardening [31]	No	Automated	Automated	Custom PHP Interpreter
WebSSARI [20]	No	Automated	Semi-Automated	None

Table 4: Evaluation of Code Improvement Techniques with Respect to Common Development Errors.



Technique	*Input type checking*	*Encoding of input*	*Identification of all input sources*	*Positive pattern matching*
JDBC-Checker [12]	Yes	No	No	No
Java Static Tainting [23]	No	No	Yes	No
Safe Query Objects [7]	Yes	Yes	N/A	No
SecuriFly [26]	No	Yes	Yes	No
Security Gateway [26]	Yes	Yes	No	Yes
SQL DOM [27]	Yes	Yes	N/A	No
WebSSARI [20]	Yes	Yes	Yes	Yes

6. References

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

[2] C. Anley. (more) Advanced SQL Injection. White paper, Next Generation Security Software Ltd., 2002.

[3] D. Aucsmith. Creating and Maintaining Software that Resists Malicious Attack. http://www.gtisc.gatech.edu/bio_aucsmith.html, September 2004. Distinguished Lecture Series.

[4] F. Bouma. Stored Procedures are Bad, O'kay? Technical report, Asp.Net Weblogs, November 2003. http://weblogs.asp. net/fbouma/archive/2003/11/18/38178.aspx.

[5] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, pages 292–302, June 2004.

[6] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In International Workshop on Software Engineering and Middleware (SEM), 2005.

[7] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), 2005.

[8] M. Dornseif. Common Failures in Internet Applications, May 2005. http://md.hudora.de/presentations/ 2005-common-failures/ dornseif-common-failures-2005-05-25.pdf.

[9] E. M. Fayo. Advanced SQL Injection in Oracle Databases. Technical report, Argeniss Information Security, Black Hat Briefings, Black Hat USA, 2005.

[10] P. Finnigan. SQL Injection and Oracle - Parts 1 & 2. Technical Report, Security Focus, November 2002. http://securityfocus.com/infocus/1644, http://securityfocus.com/infocus/1646.

[11] T. O. Foundation. Top Ten Most Critical Web Application Vulnerabilities, 2005. http: //www.owasp.org/documentation/topten.html.

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

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

[14] N. W. Group. RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1. Request for comments, The Internet Society, 1999.

[15] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In Proceedings 21st Annual Computer Security Applications Conference, Dec. 2005.

[16] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In Proceedings of the IEEE and ACM International

Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.

[17] W. G. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005), pages 22–28, St. Louis, MO, USA, May 2005.

[18] M. Howard and D. LeBlanc. Writing Secure Code. Microsoft Press, Redmond, Washington, second edition, 2003.

[19] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In Proceedings of the 11th International World Wide Web Conference (WWW 03), May 2003.

[20] Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In Proceedings of the 12th International World Wide Web Conference (WWW 04), May 2004.

[21] S. Labs. SQL Injection. White paper, SPI Dynamics, Inc., 2002. http://www.spidynamics.com/assets/documents/ WhitepaperSQLInjection.pdf.

[22] D. Litchfield. Web Application Disassembly with ODBC Error Messages. Technical document, @Stake, Inc., 2002. http://www.nextgenss.com/papers/webappdis.doc.

[23] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In Proceedings of the 14th Usenix Security Symposium, pages 271–286, Aug. 2005.

[24] C. A. Mackay. SQL Injection Attacks and Some Tips on How to Prevent Them. Technical report, The Code Project, January 2005. http://www.codeproject.com/cs/database/ SqlInjectionAttacks.asp.

[25] O. Maor and A. Shulman. SQL Injection Signatures Evasion. White paper, Imperva, April 2004. http://www.imperva.com/ application defense center/white papers/ sql injection signatures evasion.html.

[26] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA 2005), pages 365–383, 2005.

[27] R. McClure and I. Kr¨uger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pages 88–96, 2005.

[28] S. McDonald. SQL Injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity.org, April 2002. http://www.governmentsecurity.org/articles/ SQLInjectionModesofAttackDefenceandWhyIt Matters.php.

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