1. What is the paper about? What is/are the vulnerability? What causes the vulnerability?

The paper discusses SQL injection attacks, the techniques those wishing to exploit SQL injection vulnerabilities may use, and the existing detection/prevention techniques to guard against those attacks. SQL Injection attacks are considered one of the most serious threads for web applications as they may expose sensitive user information, which can lead to identify theft, fraud, and other security violations. Despite how serious the SQL Injection vulnerabilities are, they stem from a quite simple problem, insufficient validation/sanitation of user input. Developers have propose a wide range of techniques and coding guidelines to prevent these vulnerabilities, yet applications that are vulnerable to SQL injection attacks are still widespread.

1. What is/are the contributions of the paper? How was the vulnerability or insecurity discovered?

The paper discusses two main characteristics of any SQL Injection Attack: injection mechanism and attack intent. It uses these characteristics to not only give background information on SQL Injection Attacks, but to use when evaluating different attack techniques and evaluate known ways to defend/protect against the attacks.

First on the list is injection through user input. In this case, the attacker injects specially crafted user input, which will inject SQL commands. Injection can also occur through cookies. The cookie’s content can be used to build SQL queries and the cookie is under the client’s control, a malicious client could tamper with the cookie’s content to embed an attack in the cookie. Then if the cookie were used to restore the client’s state information, the attack would be submitted to the system. A third mechanism is injection through server variables. An attacker could create values for server variables placed in HTTP and network headers. The attack would be triggered if the server variables were logged to a database without sanitization. The paper also recognizes a final mechanism called second-order injection. In this case, the attackers would place malicious code/input into the system that will trigger a SQL injection attack when the user inputs certain data at a later time.

The paper also discusses ten main attack intents.

Attacks can also be characterized based on the goal, or *intent*, of the attacker. Therefore, each of the attack type definitions that

we provide in Section 4 includes a list of one or more of the attack intents defined in this section.

*Identifying injectable parameters:* The attacker wants to probe a Web application to discover which parameters and user-input fields are vulnerable to SQLIA.

*Performing database finger-printing:* The attacker wants to discover the type and version of database that a Web application is

using. Certain types of databases respond differently to different queries and attacks, and this information can be used to “fingerprint” the database. Knowing the type and version of the database used by a Web application allows an attacker to craft databasespecific attacks.

*Determining database schema:* To correctly extract data from a database, the attacker often needs to know database schema information, such as table names, column names, and column data types. Attacks with this intent are created to collect or infer this kind of information.

*Extracting data:* These types of attacks employ techniques that will extract data values from the database. Depending on the type

of the Web application, this information could be sensitive and highly desirable to the attacker. Attacks with this intent are the

most common type of SQLIA.

*Adding or modifying data:* The goal of these attacks is to add or change information in a database.

*Performing denial of service:* These attacks are performed to shut down the database of a Web application, thus denying service

to other users. Attacks involving locking or dropping database tables also fall under this category.

*Evading detection:* This category refers to certain attack techniques that are employed to avoid auditing and detection by system

protection mechanisms.

*Bypassing authentication:* The goal of these types of attacks is to allow the attacker to bypass database and application authentication mechanisms. Bypassing such mechanisms could allow the attacker to assume the rights and privileges associated with another application user.

*Executing remote commands:* These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.

*Performing privilege escalation:* These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attacks focus on exploiting the database user privileges.

**4. SQLIA TYPES**

In this section, we present and discuss the different kinds of SQLIAs known to date. For each attack type, we provide a descriptive *name*, one or more *attack intents*, 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 Intent*: 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. In general, for a tautology-based attack to work, an attacker must consider not only the injectable/vulnerable parameters, but also the coding constructs that evaluate the query results. Typically, the attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned.

**Illegal/Logically Incorrect Queries**

*Attack Intent*: 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 aWeb application. The attack is considered a preliminary, informationgathering 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. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Syntax errors can be used to identify injectable parameters. Type errors can be used to deduce the data types of certain columns or to extract data. Logical errors often reveal the names of the tables and columns that caused the error.

**Union Query**

*Attack Intent*: 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. With this technique, an attacker can trick the application into returning data from a table different from the one that was intended by the developer. 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.

**PiggyBacked Queries**

*Attack Intent*: 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 intended 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. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first. This type of attack can be extremely harmful. If successful, attackers can insert virtually any type of SQL command, including stored procedures,1 into the additional queries and have them executed along with the original query. 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.

**Stored Procedures**

*Attack Intent*: 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 backenddatabase 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]. 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

**Inference**

*Attack Intent*: Identifying 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. Since database error messages are unavailable to provide the attacker with feedback, attackers must use a different method of obtaining a response from the database. 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 wellknown 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.

**Alternate Encodings**

*Attack Intent*: 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. Contributing to the problem is that different layers in an application have different ways of handling alternate encodings. 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

1. The detailed techniques to solve the problem.

The TaintCheck software identifies which parts of the payload could be useful in an attack signature and performs automatic semantic analysis. At the processor-instruction level, TaintCheck monitors every byte of every attack payload is used by the vulnerable program. This analysis can be used to generate an attack signature directly instead of relying upon signatures generated from content pattern extraction. Since the semantic analysis provides information about not only the possible vulnerability, but also how the vulnerability may be exploited, less payloads would be necessary to generate an accurate signature then with content pattern extraction. This could help minimize the damage caused by a newly introduced work by catching the exploit/vulnerability much earlier and allowing defenses to be put in place to contain it.

**5. 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.

**5.1 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. As we outlined in Section 2.1, 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.

**5.2 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 [37]. 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 staticallybuiltmodels. Queries that violate themodel 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 [35] 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-validationrelated 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 [31] and Pietraszek and Berghe [32] 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.

*IntrusionDetection Systems.* Valeur and colleagues [36] 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 [33] 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.

1. What are the strength/weaknesses of the paper?

The first weakness I could see with the TaintCheck software slows server execution 1.5 to 40 times. This is not only a large range that could depend on many factors, but most programs would consider a server running at 40 times slower than normal to be unacceptable for performance standards.

The second weakness is that the TaintCheck software detects the attack at the time of use instead of the time of write. This means that the exploit could be in place and undetected for a long time before the software determines that it was designed for an attack. This puts the vulnerable system in the precarious situation where it is trying to stop an attack in progress instead of trying to prevent it from occurring in the first place.

One strength of the paper is that in their testing of the TaintCheck software they used synthetic exploits to test the programs for vulnerabilities. In this case, they could exercise threads of execution that would not normally be touched during runtime. This could help to expose possible vulnerabilities in programs before the program is delivered, released, or put into production.

*Another strength is the outline their limitations. While this may seem counterintuitive because it gives attackers a way to get around their defenses, it also gives those wishing to use their product a good general idea of where else their system may be vulnerable.*

1. What can you do better?

One improvement to TaintCheck would be to share any the signatures generated between programs. This would help other programs detect the attack and keep other vulnerable programs or systems from being infected in the first place.

In addition, I would include support for branch functions as they give the attacker a way to mask their program flow, which can allow malicious code to go undetected.