A Review on SQL Injection

Vidushi[1], Prof. S. Niranjan[2]
[1,2] Department of Computer Science &Engineering,
Ganga Institute of Technology and Management,
Kablana, Jhajjar, Haryana, India

***Abstract*—SQL Injection is a technique of introducing malicious code into entry fields. This is one of the attacking methods used by hackers to steal the information of organizations. Security of databases is still an open challenge. SQL injection is a major threat to our web application which gives the unauthorized access to sensitive information of the database to the attackers. Researchers and practitioners have proposed various methods to address the SQL injection problem, current approaches either fail to address the full scope of the problem or have limitations that prevent their use and adoption. Many researchers and practitioners are familiar with only a subset of the wide range of techniques available to attackers who are trying to take advantage of SQL injection vulnerabilities. As a consequence, many solutions proposed in the literature address only some of the issues related to SQL injection.**

Index Terms*—Introduction, History, SQL Injection,Attack Intent, Sources, Types, Detection Techniques, References.

1.INTRODUCTION

SQL injection comes with a bang and caused revolution in database attacking . In recent years, with the explosion in web-based commerce and information systems, databases have been drawing ever closer to the network and it is critical part of network security. Database is the storage brain of the website. A hacked database is the source of password and sensitive information like credit card number, bank account number and every important thing that is forbidden. SQL injection can cause severe damage to our database. Importance should be given for preventing database exploitation by SQL injection. The aim of this paper is to create awareness among web developers or database

administrators about the urgent need for database security. Our ultimate objective is to totally eradicate the whole concept of SQL injection and to avoid this technique becoming a plaything in hands of exploiters.[1] A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.[2]

II. . HISTORY OF SQL INJECTION-

Ever since the advent of the computer, there have always been people trying to hack them. William D. Mathews of MIT discovered a flaw in the Multics CTSS password file on the IBM 7094 in 1965;

John T. Draper ("Captain Crunch") discovered a cereal toy whistle could provide free phone calls around 1971; The Chaos Computer Club, the Cult of the Dead Cow, 2600, the infamous Kevin Mitnick, even computing godfather Alan Turing and his World War II German Enigma-cipher bustingBombe, all and more have participated in hacking computers for as long as computers have existed.

Through the 1980s and 1990s, the world began to see the advent of the personal computer, the internet, and the world wide web. Telephone lines in millions of homes began screaming with the ear-piercing tones of dial up connections. AOL, CompuServe, Juno, and more began providing home users with information portals and gateways to the web. The information age was born; as was the age of information security (and, indeed, *in*security).

As websites began to form by the thousands per day, so did the technology behind them. Websites went from merely being static pages of text and images to dynamic web applications of custom-tailored content. HTML, CSS, and JavaScript grew into bigger and better systems for stitching content together in the browser, and the browser itself evolved, through Internet Explorer, Netscape, Firefox, Chrome, and more. PHP and Perl CGI, among others, became the languages of choice for backend website scripting to real-time generate the HTML and other elements browsers would render. Database systems came and went, but MySQL became the most popular. In fact, a lot of things came and went -- Dot-Com bubble, anyone? -- but one thing always remained: web application security.[3]

Here is a small sampling by Mavituna Security:

- In 2012, 97% of all data breaches world wide were SQL injection attacks.
- In one month, from the end of 2011 to early 2012, over 1,000,000 sites were successfully attacked with SQL injection.

- SQL injection has remained in the top 10 list of vulnerabilities compiled by the Open Web Application Security Project.

III. A SIMPLE SQL INJECTION

The injection process works by prematurely terminating a text string and appending a new command.[3] Because the inserted command may have additional strings appended to it before it is executed, the malefactor terminates the injected string with a comment mark "--". Subsequent text is ignored at execution time.

A simple SQL injection is shown through the following script

The script builds an SQL query by concatenating hard-coded strings together with a string entered by the user:

var Shipcity;

ShipCity = Request.form ("ShipCity");

var sql = "select * from OrdersTable where ShipCity = '" + ShipCity + "'";

The user is prompted to enter the name of a city. If she enters Redmond, the query assembled by the script looks similar to the following:

SELECT * FROM OrdersTable WHERE ShipCity = 'Redmond'

However, assume that the user enters the following:

Redmond'; drop table OrdersTable--

In this case, the following query is assembled by the script:

SELECT * FROM OrdersTable WHERE ShipCity = 'Redmond';drop table OrdersTable--'

The semicolon (;) denotes the end of one query and the start of another. The double hyphen (--) indicates that the rest of the current line is a comment and should be ignored. If the modified code is syntactically correct, it will be executed by the server. When SQL Server processes this statement, SQL Server will first select all records in OrdersTable where ShipCity is Redmond. Then, SQL Server will drop Orders Table.

As long as injected SQL code is syntactically correct, tampering cannot be detected programmatically. Therefore, you must validate all user input and carefully review code that executes constructed SQL commands in the server that you are using. Coding best practices are described in the following sections in this topic.[2]

IV. ATTACK INTENT

Attacks can also be characterized based on the goal, or intent,of the attacker. Therefore, we can define[4] several intents as follows:

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 froma 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 thatwill 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 isto 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 advantageof 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.

5. Sources[5] of SQL Injection Attack

- Injection through user input

Malicious strings are introduced in web forms through user inputs.

- Injection through cookies

Modified cookie fields contain attack strings.

- Injection through server variables

Headers are manipulated to c ontain attack strings.

5.1 Second-order injection

• Trojan horse input seems fine until used in a certain situation.

Attack does not occur when it first reaches the database, but when used later on.

Input: admin'-- ===> admin\'--

queryString ="UPDATE users SET pin=" + newPin +

" WHERE userNa me='" + userName + "' AND pin=" + oldPin;

queryString ="UPDATE users SET pin='0'

WHERE userName= 'admin'--' AND pin=1";

6. Types of SQL Injection

- Piggy-backed Queries

Attack Intent: Extraction, modify datasets, execute remote commands, DoS

Different than other attacks not only because hacker attempts to execute two commands at once but also due to the first query not intended to modify or cause damage. First query is valid and runs normally but when delimiter is recognized DBMS executes second malicious query
System that is vulnerable to piggy-backed queries is generally due to misconfiguartion which allows for multiple statements in one query

• Tautologies

This attack works by inserting an "always true" fragment into a WHERE clause of the SQL statement[7]. This is often used in combination with the insertion of a double dash -- to cause the remainder of a statement to be ignored, ensuring extraction of largest amount of data. Tautological injections can include techniques to further mask SQL expression fragments, such as the following:

' or 'simple' like 'sim%' --

' or 'simple' like 'sim' || 'ple' --

The || in the example is used to concatenate strings, when evaluated the text 'sim' || 'ple'becomes 'simple'.

• Alternate Encodings

In this case, text is encoded to avoid detection by defensive coding practices. It can also be very difficult to generate rules for a WAF to detect encoded input. Encodings, in fact, can be used in combination with other attack classifications. Since databases parse comments out of an SQL statement prior to processing it, comments are often used in the middle of an attack to hide the attack's pattern. Scanning and detection techniques, including those used in WAFs, have not been effective against alternate encodings or comment based obfuscation because all possible encodings must be considered.

• Stored Procedure Attacks: These attacks attempt to execute database stored procedures. The attacker initially determines the database type (typically through illegal/logically incorrect queries) and then uses that knowledge to determine what stored procedures might exist. Contrary to popular belief, using stored procedures does not make the database invulnerable to SQL injection attacks. Stored procedures can be vulnerable to privilege escalation, buffer overflows, and even provide administrative access to the operating system.
• Illegal/Logically Incorrect Queries : Attackers use this approach to gather important information about the type of database and its structure. Attacks of this nature are often used in the initial reconnaissance phase of the attack to gather critical knowledge used in subsequent attacks. Returned error pages that are not filtered can be very instructive. Even if the application sanitizes error messages, the fact that an error is returned or not returned can reveal vulnerable or injectable parameters. Syntax errors identify injectable parameters; type errors help decipher data types of certain columns; logical errors, if returned to the user, can reveal table or column names.

The specific attacks within this class are largely the same as those used in a Tautological attack. The difference is that these are intended to determine how the system responds to different attacks by looking at the response to a normal input, an input with a logically true statement appended (typical

tautological attack), an input with a logically false statement appended (to catch the response to failure) and an invalid statement to see how the system responds to bad SQL. This will often allow the attacker to see if an attack got through to the database even if the application does not allow the output from that statement to be displayed.

• Union Query: This attack exploits a vulnerable parameter by injecting a statement of the form:

foo'UNION SELECT <rest of injected query>

The attacker can insert any appropriate query to retrieve information from a table different from the one that was the target of the original statement.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.

V. PREVENTION TECHNIQUES

• Defensive Coding Best Practices

The root cause of SQL injection vulnerabilities is insufficient input validation.

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 , there

• Penetration Testing

• Static Analysis of Code

• Safe Development Libraries

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 [10, 11,12]. 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" [9]. 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 mayrefer to [30,31,29,16] for examples of how these pseudo-remedies can be subverted.

VIII. 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 [8] propose WAVES, 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 [28,29]. 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 [27]. 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 [26,25]. 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 staticallybuilt 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 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 [14] and SQLCheck [15] 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 educed 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 attacke.rs 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 [10]. 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.

For many types of functions and applications, this assumption is too strong. Livshits and Lam [11] 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 untain nting 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 [22] and Pietraszek and Berghe [23] 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 [20] and SecuriFly [21] 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.

New Query Development Paradigms. Two recent approaches, SQL DOM [18] and Safe Query Objects [24], 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 [17] 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 [12] 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 [29] 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.

VIII. CONCLUSION

In this paper, I have presented a survey and comparison of current techniques for detecting and preventing SQLIAs. To perform this evaluation, I first identified the various types of SQLIAs known to date. I then evaluated the considered techniques in terms of their ability to detect and/or prevent such attacks. I 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, I 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 genera al detection and prevention techniques. Future evaluation work should focus on evaluating the techniques's precision and effectiveness in practice.

REFERENCES

[1] http://www.jiclt.com/index.php/jiclt/article/view/141[2]
[2] https://technet.microsoft.com/en-us/library/ms161953(v=SQL.105).aspx
[3] https://www.netsparker.com/blog/web-security/sql-injection-vulnerability-history/
[4] Halfond,viegas,orso "A Classification of SQL, Injection Attacks and Countermeasures", ISSSE , 2006
[5] William Halfond – ISSSE 2006 – March 14th, 2006
[6]M. Howard and D. LeBlanc. Writing Secure Code. Microsoft Press, Redmond, Washington, second edition, 2003.
[7] http://www.dbnetworks.com/pdf/sql-injection-detection- eb-environment.pdf
[8] 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.
[9] M. Howard and D. LeBlanc. Writing Secure Code. Microsoft Press, Redmond, Washington, second edition, 2003.
[10]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.
[11] 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
[12] D. Scott and R. Sharp. Abstracting Application-level Web Security. In Proceedings of the 11th International Conference on the World Wide Web (WWW 2002), pages 396–407, 2002.
[13] E. M. Fayo. Advanced SQL Injection in Oracle Databases. Technical report, Argeniss Information Security, Black Hat Briefings, Black Hat USA, 2005.
[14]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
[15]Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), Jan. 2006.
[16]S. McDonald. SQL Injection Walkthrough. White paper, SecuriTeam, May 2002. http://www.securiteam.com/securityreviews/5DP0N1P76E.html.
[17] 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.
[18] R. McClure and I. Kruger. 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
[19]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.
[20]V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In Proceedings 21st Annual Computer Security Applications Conference, Dec. 2005.
[21] 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.
[22]A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting Information. In Twentieth IFIP International Information Security Conference (SEC 2005), May 2005.
[23]T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In Proceedings of Recent Advances in Intrusion Detection (RAID2005), 2005.
[24] 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.
[25] 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. To appear.
[26]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.
[27] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004), pages 70–78, 2004.
[28]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.
[29] 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.
[29] S. McDonald. SQL Injection: Modes of attack, defense, and why it matters. White paper, Government Security.org, April 2002.
[30] C. Anley. Advanced SQL Injection In SQL Server Applications. White paper, Next Generation Security Software Ltd., 2002.
[31] 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.