SQL-IDS: A Specification-based Approach for SQL-Injection Detection

Konstantinos Kemalis and Theodoros Tzouramanis
Department of Information & Communication Systems Engineering,
University of the Aegean,
Karlovassi, Samos, 83200, Greece
Tel.: +30-22730-82253
{kkemalis, ttzouram}@aegean.gr

ABSTRACT

Vulnerabilities in web applications allow malicious users to obtain unrestricted access to private and confidential information. SQL injection attacks rank at the top of the list of threats directed at any database-driven application written for the Web. An attacker can take advantages of web application programming security flaws and pass unexpected malicious SQL statements through a web application for execution by the back-end database. This paper proposes a novel specification-based methodology for the detection of exploitations of SQL injection vulnerabilities. The new approach on the one hand utilizes specifications that define the intended syntactic structure of SQL queries that are produced and executed by the web application and on the other hand monitors the application for executing queries that are in violation of the specification.

The three most important advantages of the new approach against existing analogous mechanisms are that, first, it prevents all forms of SQL injection attacks; second, its effectiveness is independent of any particular target system, application environment, or DBMS; and, third, there is no need to modify the source code of existing web applications to apply the new protection scheme to them.

We developed a prototype SQL injection detection system (SQL-IDS) that implements the proposed algorithm. The system monitors Java-based applications and detects SQL injection attacks in real time. We report some preliminary experimental results over several SQL injection attacks that show that the proposed query-specific detection allows the system to perform focused analysis at negligible computational overhead without producing false positives or false negatives. Therefore, the new approach is very efficient in practice.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – Reliability, Validation; D.3.1 [**Programming Languages**]: Formal Definitions and Theory – Syntax; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems – *Grammar types, Parsing*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection – *Unauthorized access.*

General Terms

Languages, Security, Verification, Experimentation.

Keywords

Database security, world-wide web, web application security, SQL injection attacks, specification-based runtime validation.

1. INTRODUCTION

In the recent years, web applications have tended to become commonplace. Nowadays there is a plethora of web applications that cover a wide range of daily needs. A large number of electronic transactions, including e-commerce, e-banking, e-voting, e-learning, and e-health among others, can be conducted online at any time and from any place [12]. However, in all these Internet applications exposed to hacking attempts, security-related problems are a major issue.

SQL injection represents today the most common indirect attack technique against web-powered databases and can disassemble effectively the secrecy, integrity and availability of web applications. SQL injection occurs when an attacker inserts malicious SQL code into an SQL query by manipulating data input into an application. This kind of vulnerability is a serious threat to any web application that reads input from users and uses it to build and execute SQL queries to an underlying database. With SQL injection, the attacker can run arbitrary SQL queries, extracting sensitive customer and order information from e-commerce applications, or s/he can bypass strong security mechanisms compromising the back-end databases and the data server file system.

In the face of these threats, a surprisingly high number of systems on the Internet are entirely vulnerable to such attacks, leaving even experienced professional programmers unable to cover all possible SQL injection techniques. Common software that has been found to be susceptible to SQL injection includes PHPNuke, vBulletin, WordPress, WBBlog, and literally hundreds of others. The most famous SQL injection incident that has occurred probably took place in early 2005 when the CardSystems Solutions database was found to be compromised [5]. Over a quarter of a million credit cards numbers were stolen and over ten million dollars of fraudulent activity were involved in this case.

In order to perform SQL injection hacking, all an attacker needs is a web browser and a degree of guess work to find important table and field database names, which explains why SQL injection is one of the most common application layer attacks currently being used on the Internet. An Open Web Application Security Project Foundation (OWASP Foundation) classification places SQL injection attacks in the second place on their list of the ten most critical web application security vulnerabilities [17].

Although there has recently been a great deal of attention devoted to the problem of SQL injection vulnerabilities [3, 4, 6, 7, 15, 16, 19, 20, 21], many proposed solutions fail to address all types of SQL injection attacks. Researchers and practitioners are often unaware of the myriad of different techniques that can be used to perform SQL injection attacks implying that solutions proposed protect only from a subset of all possible attacks. In addition, many of the proposed schemes require a (sometimes extensive) modification of the web application programming scripts in order to be able to protect the underlying database.

In this article, we propose a new approach for the prevention of any form of SQL injection attack. The basic idea of the novel methodology consists in writing specifications for the web application that describe the intended structure of SQL statements that are produced by the application, and in automatically monitoring the execution of these SQL statements for violations with respect to these specifications.

Another important feature of the proposed technique rests in the fact that there is no need to modify the source code of existing web applications to protect them from SQL injection attacks. Therefore, this approach is a protection mechanism that is independent of any particular application environment or DBMS and it can be deployed without having to perform costly and time-consuming tuning and configuration.

We also design and implement a prototype defense tool for the detection of SQL injection attacks, called SQL Injection Detection System (SQL-IDS). This system implements the proposed specification-based detection methodology, and it is applied to counter attacks on Java-based web applications, although it can straightforwardly be applied to web applications written in different languages. We report some of our preliminary experimental results over several SQL injection attacks which indicate that the new technique is effective, that it involves negligible runtime overhead and functions completely transparent to the developer.

This article will unfold along these lines: Section 2, which follows, describes the way in which SQL can be 'injected' into a web-powered database. Section 3 describes the proposed detection methodology for SQL injection attacks. Section 4 describes the design and implementation of a prototype specification-based detection system. Section 5 discusses issues related to the evaluation of the proposed scheme. Section 6 reviews related work and, finally, Section 7 concludes and presents suggestions for further research.

2. SQL INJECTION ATTACKS

SQL injection is a particularly insidious attack since it transcends all of the good planning that goes into a secure database setup and allows mistrusted individuals to inject code directly into the database management system (DBMS) through a vulnerable application [14]. The basic idea behind this attack is that the malicious user counterfeits the data that a web application sends to the database aiming at the modification of the SQL query that will be executed by the DBMS [18]. This falsification seems harmless at first glance but it is actually exceptionally vicious. One of the most worrying aspects of the problem is that successful SQL injection is very easy to perform, even if the developers of the web applications are aware of this type of attack.

The technologies vulnerable to SQL injection attack are dynamic script languages like ASP, ASP.NET, PHP, JSP, CGI, etc [2]. Let's imagine, for example, the typical user and password entry form of a web application that appears in Figure 1. When the user provides her/his credentials, an ASP (Active Server Page) code similar to the one that appears in Figure 2 might undertake to produce the SQL query that will certify the user's identity.

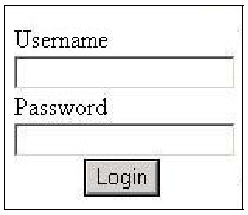

Figure 1: A typical user authentication form in a web application.

```
username = Request.form("username");
        password = Request.form("password");
        var con = Server.CreateObject(ADODB.Connection");
        var rso = Server.CreateObject(ADODB.Recordset");
        var sql = "SELECT * FROM users
        WHERE username = ' " + username +" ' AND
               password = ' " + password + " ' ";
        rso.open(sql, con);
        if not rso.eof () then
                response.write ("Welcome to the database!");
```

Figure 2: An ASP code example that manages the users' login requests in a database through a web application.

In practice, when the user types a combination of valid login name and password the application will confirm the elements by submitting a relative SQL query in some table *USERS* with two columns: the column *username* and the column *password*. The most important part of the code of Figure 2 is the line:

*sql = "SELECT * FROM users WHERE username = ' " + username +" ' AND password = ' " + password + " ' ";*

The query is sent for execution into the database. The values of the variables *username* and *password* are provided by the user. For example, if the user types:

username: *george*
password: *45dc&vg3*

the SQL query that is produced is the:

*SELECT * FROM users WHERE username = 'george' AND password = '45dc&vg3';*

which means that if this pair of *username* and *password* is stored in the table *USERS*, the authentication is successful and the user is inserted in the private area of the web application. If however the malicious user types in the entry form the following unexpected values:

username: *george*

password: *anything' OR '1' = '1*

then the dynamic SQL query is the:

*SELECT * FROM users WHERE username = 'george' AND password = 'anything' OR '1' = '1';*

The expression *'1'='1'* is always true for every row in the table, and a true expression connected with '*OR*' to another expression will always return true. Consequently, the database returns all the tuples of the table *USERS*. Then, provided that the web application received, for an answer, certain tuples, it concludes that the user's password is '*anything*' and permits his/her entry. In the worst case the web application presents on the screen of the malicious user all the tuples of the table *USERS*, which is to say all the *usernames* with their *passwords*.

If the malicious user knows the whole or part of the login name of a user, s/he can log on as he/she, without knowing his/her *password*, by entering a *username* like in the following form:

username: *' OR username LIKE 'admin%'--*

password:

The '--' sequence begins a single-line comment in Transact-SQL, so in a Microsoft SQL Server environment everything after that point in the query will be ignored. By similar expressions the malicious user can change a user's *password*, drop the *USERS* table, create a new database: s/he can effectively do anything s/he can express as an SQL query that the web application has the privilege of doing, including running arbitrary commands, creating and running DLLs within the DBMS process, shutting down the database server or sending all the data off to some server out on the Internet.

3. SPECIFICATION-BASED DETECTION

This section proposes a novel methodology to detect exploitations of SQL injection vulnerabilities in web applications. The new approach utilizes security specifications that describe the intended syntactic structure of SQL statements that are produced by the application. SQL statements that do not conform to the specifications are considered as security violations and their execution is blocked.

The detection technique is based on the assumption that injected SQL commands have differences in their structure with regard to the expected SQL commands that are built by the scripts of the web application. Therefore, if the intended structure of the expected SQL commands has been explicitly pre-determined, it is possible to detect malicious modifications that alter this structure.

The new methodology consists of a set of phases that should be followed, so that each SQL statement to be analyzed and checked in order to make sure that it has not been poisoned via the injection of malicious code. The phases of the methodology are described in greater detail below.

Phase 1: Definition of specifications

Of high importance for the proposed technique are the specifications of the web application. Specifications are a set of rules that describe the expected structure of SQL statements that are produced by the application. For each original SQL statement that is expected to be executed to the back-end database, a rule is created that defines its syntactic structure.

Phase 2: Interception of SQL statements

The traffic between the web application server and the data server is filtered and the SQL statements that are sent from the application are not transmitted directly for execution to the back-end database. Instead of this, each SQL statement passes through a validation process that checks it for the potential existence of SQL injection poisoning attacks. The detection and validation processes are described in more detail in the following phases.

Phase 3: Lexical analysis

Each intercepted SQL statement initially is recognized as an arbitrary set of characters. The next phase is to determine the syntactic units by which the SQL statement is composed. For this reason, this arbitrary input passes through a lexical analysis process in which the characters are grouped into tokens (i.e. logic units) that consist of one or more characters which may represent key-words of SQL language (eg. *SELECT*, *DELETE*, *OR*, *AND*), symbols (eg. +, -, <, <=), constants (eg. 123, 3.1416), variables, etc..

Phase 4: Syntactical verification of SQL statements

This is the main part of the detection and validation processes. The sequence of tokens that is produced for each SQL statement in the previous phase is checked for its syntactical correctness. An SQL statement is considered to be valid if it does not violate the syntactical rules that exist in the corresponding specification, as these have been pre-defined in the first phase of the methodology.

Phase 5: Forwarding valid SQL statements to the database

If the syntactical validation process confirms that the SQL statement satisfies the rules of the pre-defined specification (and thus, it does not contain injected code), the SQL statement is forwarded to the database server for execution.

Phase 6: Logging

By the completion of the validation process for each SQL statement, essential information with sufficient detail is audited in a log file, with the intention to facilitate the administrators' task with all the useful information that they need to inspect the security of the web application. Log files may also help developers to improve their web application performance and the effectiveness and the precision of its security on-the-fly, for example by writing more strictly-defined SQL-statement specifications.

4. DESIGN AND IMPLEMENTATION

This section presents the design and implementation of a prototype specification-based detection system, the SQL-Injection Detection System (SQL-IDS), which is based on the proposed methodology and in Section 5 it is applied on Java-based web applications to verify the efficiency of the new methodology. As was mentioned earlier, the goal of SQL-IDS is to provide a protection mechanism that is independent of any particular target system, application environment, or DBMS.

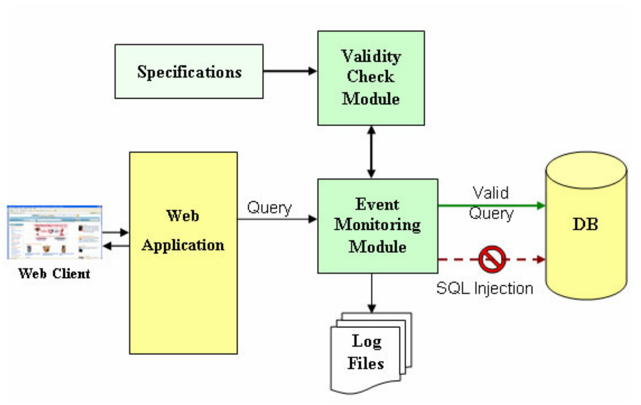

Figure 3: The architecture of SQL-IDS.

Figure 3 illustrate the structure of the SQL-IDS prototype. It consists of three main components which interact with each other. These three modules are described in the following.

Event Monitoring Module (EMM): It deals with the process of SQL query interception. Each SQL query that is intercepted is sent to the Validity Check Module (which will be described next) to confirm its correctness with respect to the specifications that have been pre-defined. If the Validity Check Module certifies that the SQL query does match to the specifications, the EMM forwards the valid query to the database server to be executed regularly. In the opposite case, if it is recognized that the SQL query violates the specifications, then the EMM marks the query as a potential SQL injection attack, it prevents its execution and records essential information that concerns the attack, for the detailed examination and analysis of the attacks by the application developers. Consequently, the EMM implements the second, the fifth and the sixth phase of the proposed methodology. For the development of this module, a proxy JDBC driver is created.

Validity Check Module (VCM): It is responsible for the comparison of the structure of the SQL query with the intended structure which is described by the pre-defined specification. If the query is conformed to the specification, then it is considered to be valid and a suitable message is sent to the EMM that declare its correctness. Otherwise, if the SQL query does not match with a rule of the specification, a conclusion is derived that an SQL injection attack has modified its structure. In this case, a message is returned to the EMM that declares the detection of an SQL injection attack, and the execution of the query is prevented.

The VCM receives the SQL query as a sequence of characters and utilizes a lexical analyzer and a parser, implementing respectively for the third and fourth phase of the proposed methodology, in order to decode this sequence of characters, to recognize the structure of the SQL query and to examine the syntactic correctness of its structure. The VCM is implemented using JavaCC (Java Compiler Compiler) [13], which is a popular lexical analyzer generator and parser generator for Java language.

Specifications: A critical component of the system is the SQL statements' specifications of the web application. Via the specifications, rules are defined that describe the expected syntactic structure that an SQL query should follow in order to be considered valid. The main characteristic of the specifications is *completeness*, which means that they should include explicit rules for each statement that is expected to be executed according to the source code of the application, while each rule should cover all likely values that the SQL statement may have. If the specifications present any lack or errors, false positives or false negatives may be introduced. The syntax of the specifications needs to follow the Extended Backus-Naur Form (EBNF) notation [22], a metasyntax notation which is used widely to express context-free grammars. Specifications for SQL language standards [8, 9, 10] can be easily created based on syntactical rules that are analytically specified in EBNF grammar.

Consider for example the following SQL statement:

SELECT user_id, user_level

FROM USERS

WHERE username = 'george' AND password = '45dc&vg3';

Figure 4 shows the specification for this statement. The rules of this specification indicate the legitimate sequences of token kinds in an attack free input.

```
<Query specification>  :=
 SELECT <Select List> <From Clause> <Where Clause>

<Select List>  :=
 <Table Column> (<COMMA> <Table Column>)*

<From Clause> :=
 FROM <Table reference>

<Where Clause> :=
 WHERE <search condition> AND <search condition>

<search condition> :=
 <Table Column> "=" <STRING LITERAL>
```

Figure 4: An example of SQL statement specification.

5. EVALUATION

In the following the results obtained from the experimental evaluation of SQL-IDS are presented and discussed. The benchmarking environment is constituted by an AMD Athlon 1GHz, with 256 MB RAM and Microsoft Windows 2000. The server has been configured by Apache Tomcat (ver. 5.5.23) and Microsoft SQL Server 2000.

The evaluation of SQL-IDS is based on three criteria: the performance; the effectiveness; and the precision of the implemented system.

5.1 Performance

We measured the performance of the SQL-IDS in terms of time overhead that is introduced into the database query execution cost by the new protection technique. For the measurement, a set of 2,450 SQL queries was sent to the data server through a sample book store web application, 420 of which were poisoned with SQL injection attacks and 2,030 were attack free SQL statements.

In the experiments, it was discovered that the performance penalty that was introduced in the operation of the web application ranges from about 0 to 20 milliseconds per SQL query. The average time overhead is estimated to be less than one millisecond per query. Since the average response time for most web applications is usually a few seconds, depending on the purpose of the application, the time overhead that is introduced contributes insignificantly into the query execution, cost in the majority of the cases.

Table 1: Performance results.

SQL Queries	min time overhead	max time overhead	average time overhead
2,450	~ 0 ms	20 ms	0.5 ms

5.2 Effectiveness

The number of false negatives is critical for the level of effectiveness of the detection system. For the measurement of effectiveness an attempt was made to send to the database the set of 420 attack scenarios that was prepared in the previous experiment. Table 2 summarizes the result of the attacks. It appeared that SQL-IDS manage to detect all the attempts of attack that were injected through the application.

We have to notice that all these attacks were detected by SQL-IDS without encoding any attack-specific information into the specifications. In other words, all these attacks were "*unknown*" to the detection system. The non-existence of false negatives supports the claim that the proposed specification-based approach can be effective for the detection of novel attack vectors.

Table 2: Effectiveness results.

Attacks Performed	Attacks Detected	False Negative Rate
420	420	0 %

5.3 Precision

To make SQL-IDS reliable, the probability of falsely recognizing a valid query as an attack must be low. Precision measures the false positive rate.

To evaluate the system's precision, the set of 2,030 attack free SQL queries that was produced in the first experiment was used. As Table 3 indicates, no false alarms were reported by the new system.

Table 3: Precision results.

Attack Free SQL Queries	Attacks Detected	False Positive Rate
2,030	0	0 %

6. RELATED WORK

Various techniques have been proposed for the confrontation of the threat of SQL injection attacks. In this section, the characteristics of the best known techniques are briefly discussed and their principal weaknesses are highlighted.

6.1 Static analysis

Wassermann and Su [21] propose a static analysis framework to filter user inputs. According to them, their approach has some limitations concerning implementation-related issues, such as the way it handles some operators. Additionally, this approach is limited to discover only tautology-based attacks, i.e. attacks that always result in true or false SQL statements.

Another static analysis approach has been proposed by Livshits and Lam [15]. In this work, vulnerability patterns are described in a program query language called PQL. Static analysis is applied to find potential violations matching a vulnerability pattern. The main limitation of the method is that it cannot detect SQL injection attacks patterns that are not known beforehand, and explicitly described in the specifications.

In Huang et al. [7], preconditions are specified for all sensitive PHP functions and user input is checked against these preconditions. Like all static analysis approaches, their WebSSARI technique does not provide an automated mechanism for detection and prevention of SQL injection attacks. This is the most important reason why all the approaches of this subsection generate a significant number of false negatives.

6.2 Dynamic analysis

AMNESIA [6] uses a model-based approach to detect illegal queries before their execution into the database. In its static part, the technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, the technique uses runtime monitoring to inspect the dynamically-generated queries and check them against the statically-built model. A primary assumption regarding the applications which the method targets is that the application developer creates queries by combining hard-coded strings and variables using operations such as concatenation, appending and insertion. The main drawback of AMNESIA is that it requires the modification of the web application's source code for the successful collaboration with the security monitor officer.

SQLGuard [4] is based on comparing, at run time, the parse tree of the SQL statement before the inclusion of the user input with that resulting from parse tree of the SQL statement after the inclusion of the user input. A secret key is used for wrapping the user input, so if an attacker compromises this key, SQLGuard is difficult to prevent an attack. According to [4], the overhead to database query costs is about 3 msec (the characteristics of their server were very similar to ours: a 733MHz Windows 2000 Server machine with 256MB RAM). Another drawback of their method is that, similarly to AMNESIA, it requires the modification of the application's scripts.

SQLCheck [19] is a similar approach. It adds a key at the beginning and at the end of each user's input. At runtime, the "augmented" queries that are not in a valid syntactic form are considered attacks. The detection ability of the approach depends on the strength of the key also.

Nguyen-Tuong et al. [16] proposes a technique that is based on precisely tracking taintedness of data and checking for dangerous content. This technique requires also the modification of the execution environment.

Valeur et al. [20] developed an anomaly-based detection system that learns the profiles of the expected database access performed by web-based applications using a number of different models. Anomalous behaviour is characterized attack. However, this system and every anomaly-based detection system present a high rate of false alarms.

In SQLrand [3], the SQL standard keywords are manipulated by appending a random integer to them that an attacker cannot easily guess. Therefore, any malicious user attempting an SQL injection attack would be thwarted. This technique uses a key to randomize SQL queries, so if an attacker compromises this key, SQLrand is difficult to prevent an attack.

7. CONCLUSIONS AND FUTURE WORK

This article presents a novel methodology for the detection of SQL injection attacks. According to the approach, a specification describes the intended structure of the SQL queries that are produced by the web application. An SQL query is considered a security violation if it does not conform to the pre-defined query-specification rules.

We develop a prototype detection system that monitors Java-based applications to detect exploitations of SQL injection vulnerabilities. The detection system does not demand any change to the web application or the database schema. Our preliminary experimental results indicate that the new, automated protection solution is very effective and efficient in detecting SQL injection attacks.

In the future, we plan to carry out more experiments to verify that -as it might be expected- the behavior of the new method is not affected by the application scenario and characteristics. An issue that also requires extensive future research is the comparison of the proposed method against existing detection methods under a common and flexible benchmarking environment.

Finally, attempts should be made to modulate the proposed technique so that it might detect other types of code injection attacks and other types of attacks that offend web applications and on-line databases, such as cross-site scripting (XSS) attacks [11].

8. REFERENCES

[1] C. Andrews, D. Litchfield, B. Grindlay and NGS Software: *SQL Server Security*, McGraw-Hill/Osborne, 2003.

[2] V. Anupam and A. Mayer: “Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies”, In *Proceedings of the 7th USENIX Security Symposium,* pp. 187-200, 1998.

[3] S. Boyd and A. Keromytis: “SQLrand: Preventing SQL Injection Attacks”, In *Proceeding of the 2nd International Conference on Applied Cryptography and Network Security*, China, June 2004.

[4] G. Buehrer, B. Weide, and P. Sivilotti: “Using Parse Tree Validation to Prevent SQL Injection Attacks”, In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, Lisbon, Portugal, September 2005.

[5] E. Dash: “Lost Credit Data Improperly Kept, Company Admits”, *The New York Times*, June 20, 2005.

[6] W. Halfond and A. Orso: “AMNESIA: Analysis and Monitoring for NEutralizing SQL Injection Attacks”, In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering,* Long Beach, California, November 2005.

[7] Y.W. Huang, F. Yu, C. Hang, C.H. Tsai, D. T. Lee, S.Y. Kuo: “Securing Web Application Code by Static Analysis and Runtime Protection”, In *Proceedings of the 13th international conference on World Wide Web*, New York, USA, May 2004.

[8] ISO/IEC: “Information Technology - Database Language SQL”, July 1992.

[9] ISO/IEC: “Information Technology - Database Languages - SQL - Part 2: Foundation (SQL/Foundation)”, September 1999.

[10] ISO/IEC: “Information technology - Database Languages - SQL - Part 2: Foundation (SQL/Foundation)”, August 2003.

[11] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic: “Noxes: a client-side solution for mitigating cross-site scripting attacks”, In *Proceedings of the ACM Symposium on Applied Computing*, pp.330-337, 2006.

[12] M. Khosrow-Pour (ed.): *Encyclopedia of E-Commerce, E-Government, and Mobile Commerce*, Idea Group Reference, 2006.

[13] V. Kodaganallur: “Incorporating Language Processing into Java Applications: A JavaCC Tutorial”, *IEEE Software*, Volume 21, Issue 4, pp 70-77, July-Aug. 2004.

[14] D. Litchfield: “Web Application Disassembly with ODBC Error Messages”, 2001. Address for download: `http://www. nextgenss.com/papers/webappdis.doc`

[15] B. Livshits and M. Lam: “Finding Security Errors in Java Programs with Static Analysis”, In *Proceedings of the 14th Usenix Security Symposium,* Baltimore, USA, Aug. 2005.

[16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, D. Evans: “Automatically Hardening Web Applications Using Precise Tainting”, In *Proceedings of the 20th IFIP International Information Security Conference*, Chiba, Japan, 2005.

[17] Open Web Application Security Project Foundation (OWASP Foundation): “The Ten Most Critical Web Application Security Vulnerabilities - 2007 Update”, 2007. Address for download: `http://www.owasp.org/images/c/c7/OWASP_Top_10_2007_RC1.pdf`

[18] K. Spett: “SQL Injection: Is Your Web Applications Vulnerable?”, Technical Report, SPI Dynamics Inc., 2002.

[19] Z. Su and G. Wassermann: “The Essence of Command Injection Attacks in Web Applications”, In *Proceedings of the 33rd Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 2006.

[20] 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 & Vulnerability Assessment*, Vienna, Austria, July 2005.

[21] G. Wassermann and Z. Su: “An Analysis Framework for Security in Web Applications”, In *Proceedings of the Specification and Verification of Component-Based Systems Workshop*, Newport Beach, California, October 2004.

[22] N. Wirth: “What can we do about the unnecessary diversity of notation for syntactic definitions? ”, *Communications of the ACM*, Vol. 20, Issue 11, pp. 822-823, November 1977. Address for download the International standard (ISO 14977) that defines the EBNF: `http://standards.iso.org /ittf/PubliclyAvailableStandards/s026153_ ISO_IEC_14977_1996(E).zip`