Detection and Prevention of SQL Injection Attacks on Web Applications

Yasser Fouad[†] and Khaled Elshazly[††],

[†]Lecturer of Computer Science, Faculty of Science, Suez University, Egypt.
†† Demonstrator of Computer Science, Information System Institute, Suez, Egypt.

Summary
In this era where Internet has captured the world, level of security that this Internet provides has not grown as fast as the Internet application. Internet has eased the life of human in numerous ways, but defects such as intrusions that are attached with Internet applications keep on the growth of these applications. One such intrusion is the SQL Injection Attacks (SQLIA). In web applications with the help of the internet explorer the user tries to access the information. But most of the web applications are affected by the SQL-Injection attacks. In this paper we propose a method to detect the SQL-Injection attacks. We use a filtering proxy server to prevent a SQL-Injection attack.

Keywords: SQL-Injection, fault injection, SQL poisoning, proxy server.

1. Introduction

Most of the web applications store the data in the data base and retrieve and update information as needed. These applications are highly vulnerable to many types of attacks, one of them being SQL-Injection Attacks (SQLIA). There are many measures that can be taken to prevent SQL-Injection including making sure that the users have the minimum database privileges which possible using input to validation programming techniques, suppressing error messages returned to the client, checking error logs and filtering malicious SQL statements [1]. In research and commercial products, there is evidence proving that SQL-Injection can be prevented using means not so closely related to the database and web application. These methods of approach have been developed to produce a more generic solution to a problem that requires a lot of tweaking and attention to detail at the root of the problem by using the application code and database deployment [2].

In [3], it is introduced a new approach to automatic penetration testing by leveraging it with knowledge from dynamic analysis. There are number of reported web applications vulnerabilities is increasing dramatically. Most of them result from improper or none input validation by the web application. Most existing approaches are based on the Tainted Mode vulnerability model which cannot handle inter-module vulnerabilities.

In [4], the detection model of SQL-Injection vulnerabilities and SQL-Injection mitigation framework are proposed. These approaches are based on SQL-Injection grammar to identify the SQL-Injection vulnerabilities during software development and SQL-Injection attack on web-based applications.

In [5], a novel specification-based methodology for the prevention of SQL-Injection attacks is introduced. Two most important advantages of the new approach against existing analogous mechanisms are that: first, it prevents all forms of SQL-Injection attacks; second, Current technique does not allow the user to access database directly in database server. Used innovative technique "Web Service Oriented XPATH Authentication Technique" detect and prevent SQL Injection Attacks in database the deployment of this technique is by generating functions of two filtration models that are Active Guard and Service Detector of application scripts, additionally allowing seamless integration with currently-deployed systems. This proposed technique was able to suitably classify the attacks that performed on the applications without blocking legitimate accesses to the database (i.e., the technique produced neither false positives nor false negatives). Results show that the technique represents a promising approach to countering SQLIA's and motivate further work in this direction.

In [6], the system detects and prevents SQL-Injection queries and cross scripts, and views SQL-Injection attacking reports are investigated. SQL-Injection attacking reports used to identify user who passes the unwanted queries to the web application. Used static analysis to detect and prevent SQL-Injection attacks in compile time and dynamic analysis can be used to detect and prevent injection queries in runtime. These techniques have been implemented in ASP.Net and SQL Server, and tested by conducting various experiments and prove that the web applications and database is protected from scripting and SQL-Injection queries. In [7], an efficient approach to prevent this vulnerability is studied. Suggest have been proposed solution is based on the principle of dynamic query structure validation which is done through checking query's semantics. It detects SQL-Injection by generating a benign query from the final SQL query generated by the

Manuscript received August 5, 2013
Manuscript revised August 20, 2013

application and the inputs from the users and then comparing the semantics of safe query and the SQL query. The main focus is on stored procedure attacks in which getting query structure before actual execution is difficult.

In [8], comparing the length of two Queries (Original Query and SQL-Injection related Query) and detecting the SQL-Injection attack; if there is SQL-Injection attack then by not giving the access to the database also prevent web application from SQL Injection attacks, work has been presented and implemented using PHP codes.

In this paper, we implement a model for detection and prevention of SQL-Injection attacks on web applications. We propose a method to detect the SQL-Injection. We use a filtering proxy server to prevent a SQL-Injection attack. The filtering process seems to provide a negligible overhead.

The rest of the paper is organized as follows. In Section 2, we give an example of SQL-Injection. the prevention methods are introduced in Section 3. System overview and a new method to detect the SQL-Injection attacks are proposed in Section 4 and 5 respectively. Current status and future work are given in Sections 6 and 7 respectively.

2. Example of SQL Injection

A typical SQL statement is shown in code box 1.

Select id, forename, surname from authors where forename = 'Joe' and surname = 'Bloggs'

Code Box 1: A typical SQL statement

An important point to note is that the string literals are delimited by single quotes. The user may be able to inject some SQL if the user provides the input shown as in text box 2.

Forename: 'Joe' Surname: Bloggs

Text Box 1: User input

The query string formed from the input shown in text box 1 is shown as in code box 2.

Select id, forename, surname from authors where forename = 'Joe' and surname = 'Bloggs'

Code Box 2: Resultant query

In this case, the database engine will return an error due to incorrect syntax with the SQL query. In many web languages, a critical vulnerability is the way in which the query string is created. An example is shown as in code box 3.

*var SQL = "select * from users where username = '" + **username** + "' and password = '" + **password** + "'"*

Code Box 3: Code showing a SQL injection vulnerability

If the user specifies the input shown in text box 2, the 'users' table will be deleted, denying access to the application for all users [9].

Username: *'; drop table users–*

Text Box 2: User input to delete the users table

An attack against a database using SQL-Injection could be motivated by two primary objectives:

• To steal data from a database from which the data should not normally be available, or to obtain system configuration data that would allow an attack profile to be built. One example of the latter would be obtaining all of the database password hashes so that passwords can be brute-forced.

• To gain access to an organization's host computers via the machine hosting the database [10].

3. Prevention Methods

SQL-Injection is a relatively simple technique and on the surface protecting against it should be fairly simple; however, auditing all of the source code and protecting dynamic input is not trivial, neither is reducing the permissions of all applications users in the database itself.

It is difficult to detect SQL-Injection with an audit of the SQL commands executed. A better method is to audit the errors generated when the hacker is trying to gain access to the database. These error messages can be as useful to the hacker as they are to the database administrator building up database queries and stored procedures [1].

In the last few years, SQL-Injection attacks have been on the rise [11]. Maor and Shulman outline research that has proved that suppressing error messages – going back to the "security by obscurity" approach [1] - cannot provide a real solution to application level risk but can add a measurement of protection. Security by obscurity tries to reduce the unnecessary information from being sent back to the client. Error messages can be used to determine information such as the database type and table structure [12]. Applications have still proven to be vulnerable despite all efforts to limit information returned to the client. There are a few applications that have been developed by companies in an effort to provide a solution to this problem. Some have been outlined below:

• Secure Sphere [12] uses advanced anomaly detection, event correlation, and a broad set of signature dictionaries to protect web applications and databases.

• ModSecurity is an open source intrusion detection engine for web applications, which may provide helpful tips on how to detect SQL-Injection. [2] has developed ModSecurity for Java which is a Servlet 2.3 filter that stands between a browser and the application, monitors

requests and responses as they are passing by, and intervenes when appropriate in order to prevent attacks.

There is data that shows that injection flaws has been sixth in the top ten vulnerabilities for the past two years and that 62% of web applications are vulnerable to SQL-Injection attacks. In [13] provide evidence that there has been a lot of development and research in the area of how to detect and test sites for SQL-Injection. The presentation by [14] at a Black Hat USA 2004 convention outlines automated blind SQL-Injection techniques. He mentions that string comparison is suitable for error based SQL-Injection but not blind SQL-Injection. He also mentions that there are three kinds of SQL-Injection:-

• Redirecting and reshaping a query involves inserting SQL commands into the query being sent to the database. The commands allow a direct attack on the database.

• Error message based SQL-Injection makes use of the database error messages returned to the client. The messages provide clues as to the database type and structure as well as the query structure.

• Blind SQL-Injection which involves a lot of guesswork and thus requires a larger investment in time. The attacker tries many combinations of attack and makes the next attack attempt based on their interpretation of the resulting html page output.

In [15] provides a good background into the problem of SQL-Injection. It puts the whole problem into context. The site provides explanations of the components of SQL-Injection strings and the syntax choices. The examples include SQL-Injection attacks, creating a secure data access component using Java's regular expressions.

In [16] provides concise examples of SQL-Injection and database error messages as well as methods on how to prevent SQL-Injection. The white paper by [9] covers research into SQL-Injection as it applies to Microsoft Internet Information Server/Active Server Pages/ MS SQL Server platform. It addresses some of the data validation and database lockdown issues that are related to SQL-Injection into applications. The paper provides examples of SQL-Injection attacks and gives some insight into .asp login code and query error messages used to exploit databases.

In [1] worked examples of SQL-Injection attacks in his white paper on Detecting SQL-Injection in Oracle. It focuses on detecting SQL-Injection by auditing the error message log files. It attempts to highlight the fact that during a hacking attempt, the error messages leave a trail that can help expose the vulnerabilities of the database being attacked.

In [17] SPI Dynamics presents a paper with describing SQL-Injection in general. It goes through some common SQL-Injection techniques and proposes a solution to the problem. The paper provides a list of database tables that are useful to SQL-Injection in MS SQL Server, MS Access and Oracle. It also provides examples of SQL-Injection using select, insert, union, stored procedures. The examples work with a web service that returns information to the user. This paper deals primarily with the structure of the SQL-Injection commands and guides to overcoming possible errors returned by the database. It should be noted that SQL-Injection can still occur if there is no feedback to the client. So, one could create a new valid user in a database without receiving errors and then log on.

[18] CEO of White Hat Security, Inc., in his presentation at the Black Hat Windows Security 2004 convention, outlines the challenges of scanning web application code for vulnerabilities. He points out that the scanner is restricted to looking for classes of vulnerabilities such as SQL-Injection or cross site scripting. The reason for this being that the benefit of known security issues is lost because the remote scanner does not have access to the source code.

There is no way to provide everyone with the minimum privileges necessary. Thus the paper explores some simple techniques in extracting the logging and trace data that could be used for monitoring. [1] is an extension of a two-part paper on investigating the possibilities for an Oracle database administrator to detect SQL-Injection. This paper provides many scripts on SQL-Injection and extracting logs [1].

4. System Overview

The steps in which the SQL-Injection follows can be summarized as follows:

• Analyze the structure of SQL query commands.

• Build a parser that will check allowable patterns of SQL statements.

• Construct a list of common SQL-Injection commands.

• Create a proxy server that will alert the database administrator of possible SQL-Injection commands.

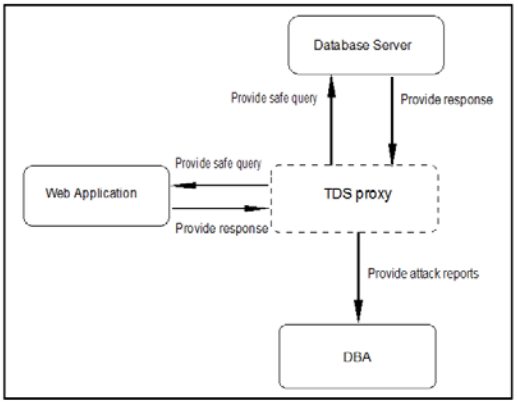

Figure 1: Information Flow Diagram

• Prevent a SQL-Injection attack to a database using this proxy server.

• Prove that SQL-Injection can be prevented using the filter developed to work on the proxy server.

• Provide sufficient logging to allow the user to isolate security holes.

• Produce a list of best practices for Database Administrators and Software Developers with respect to preventing SQL-Injection.

5. Design

Current paper aims to eliminate the possibility of SQL-Injection by the use of a proxy server, which will be placed in between the two communicating devices. This will allow for the filtering of possible SQL-Injection attempts.

The information flow diagram in figure 1 shows the flow of information between a TDSProxy server within the domain of this project and the other entities and abstractions with which it communicates. The diagram helps to discover the scope of the system and identify the system boundaries. The system under investigation (TDSProxy) is represented as a single process interacting with various data and resource flow entities via an interface. As can be seen from the diagram, the web application provides the query to TDSProxy which in turn provides safe queries to the database and attack reports to the Database Administrator. The response from the database is routed back to the web application through TDSProxy. Should the need arise, log files in the database application provide information for auditing purposes at a later stage.

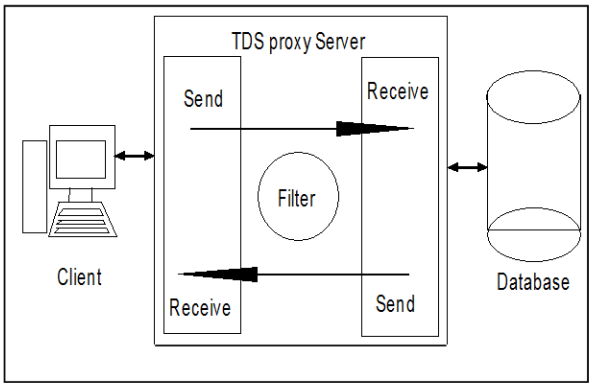

Figure 2: High Level Design View

The design and implementation steps made use of the Rational Unified Process (RUP) with the aid of Unified Mark-up Language (UML). This iterative process started off with a simple application and developed into a more complex system in subsequent iterations. The reason for using this methodology was to overcome problem areas in segments. Once the basic concept was conceived and implemented, more advanced featured were added to flesh out the software used for this proof of concept project.

The web application is where the queries are formed from the input parameters. These queries are sent to the database through TDSProxy. The bulk of the system operations take place at the TDSProxy. When the TDSProxy has filtered the query, the clean query is sent to the database server. Figure 2 illustrates how the incoming requests are filtered and only clean queries are passed on to the database for processing. For security reasons, the proxy server will sit on the same machine as the database.

The diagram in figure 2 shows all the components in the high level view of the system. The web interface is the tool used by the client to send requests to the database. The web application is pointing to TDS proxy server so that all requests and responses must go through TDSProxy. The client's web application request triggers the formation of the SQL statement which uses the input parameters of the web form to create the correct SQL statement. This SQL statement is then sent to TDSProxy. When the SQL statement is received, it is first filtered. Only clean SQL statements are then sent to the database. The database processes the request and sends its response through TDSProxy. TDSProxy in turn sends the response to the web application for processing to produce the correct view for the client.

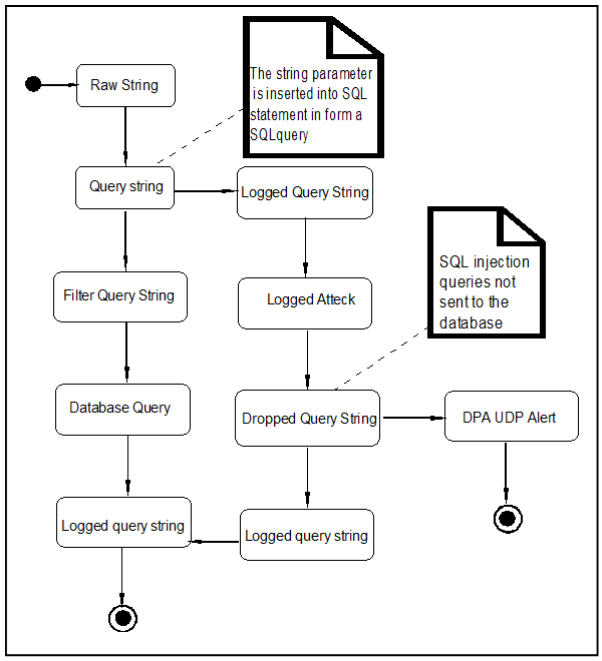

Figure 3: Flowchart of the TDS Proxy server

The flowchart in figure 3 focuses on the internally driven processes as opposed to external events. The action states in the diagram represent the decisions and behavior of the processing. Figure 3 captures the actions performed at system start-up and run time.

TDSProxy loads a configuration file at start-up. This file contains, filter settings and options as well as the settings required for the passing of data to the correct destination. Once the system has started, it is able to start receiving data from the client. When data is received from the client, the payload is analyzed. If the payload contains a SQL query, the query is logged and then filtered. If the filter process finds that there is a potential attack, the attack is logged. After logging the attack, the attack information is sent via UDP to the DBA. A false query is sent to the database and the response returned to the client. If the filter process did not pick up an attack, the query is sent to the database and the database response is returned to the client. If the payload does not contain a query, the data is simply passed on to the database.

The operations and methods of the system transform the query from one state to another depending on what route the information is flowing. These changes are shown in figure 4.

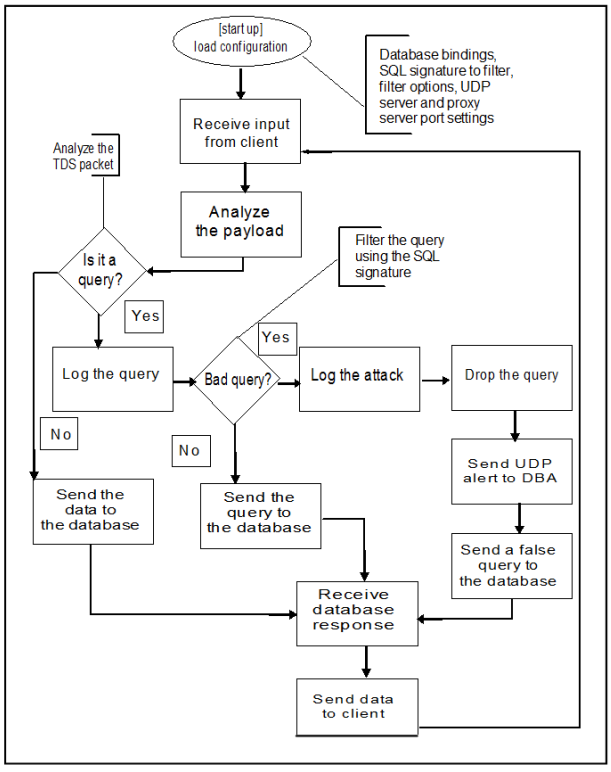

Figure 4: State Change Diagram for Client Query

The raw string becomes part of the query string through processing at the client interface. This happens when the input parameters are selected from the client interface and inserted into the hard coded query. The query may be formed at the client side or the parameters may be passed to the web application server.

Once the SQL query has been formed, it is sent to TDSProxy where it is analyzed for SQL-Injection. The query is logged and then filtered for SQL-Injection. If the query contains SQL-Injection, the attack is logged, the dangerous SQL is discarded, the DBA is notified via a UDP alert and a false query is sent to the database. The database response is then relayed to the client.

If the filtered query does not contain SQL-Injection, the query becomes a database query and is sent to the database. The database response is then relayed the client interface through TDSProxy.

6. Current Status

The implementation was done iteratively, starting off with an application that piped text (the TCP payload) through a proxy server. This was tested using a powerful networking tool called NetCat [19].

The proxy server was then improved to connect to the database using a connection string. The proxy server has a variety of potential purposes, including:

• To keep machines behind it anonymous, mainly for security.

• To speed up access to resources (using caching). Web proxies are commonly used to cache web pages from a web server.

• To prevent downloading the same content multiple times (and save bandwidth).

• To allow the browser to make web requests to externally hosted content on behalf of a website when cross-domain restrictions (in place to protect websites from the likes of data theft) prohibit the browser from directly accessing the outside domains.

There was a problem at this stage of the development. The default setup configuration of MS SQL Server 2008 allows Windows authentication only. This needed to be changed to windows and SQL server authentication in order to overcome the login error. It was thought that there was something wrong with the code when in actual fact, it was a database setting. With the correct username, password and rights, the database was manipulated by entering the SQL text on a NetCat client instance.

The next step involved sending hard coded SQL queries to the database at startup of the application. This confirmed that the username, password and privileges were correct.

An attempt at using a Microsoft Access XP data access page as the client was unsuccessful and produced many login errors. When setting up the data access page, Access

XP only accepted the use of an actual machine name and not its IP address.

For testing purposes, a direct connection to the database was set up and querying the database through the data access page was possible. The next step was to be able to route the login through TDSProxy so that the database would 'think' it was talking to an Access data access page. However, when trying to connect to the database from the data access page through the proxy server, there was a problem with the connection string. The database kept returning an error message saying that the connection was refused because it was not associated with a trusted database.

This problem was overcome by hard coding "trusted server = true" into the data access page's connection string. The login errors continued. The database kept sending back reset packets. There was no apparent reason for it not being able to log on after the data was being routed. The packet data was altered so that the source and destination ports and IP addresses made TDSProxy seem totally transparent. The first three login packets were forged from a successful login without TDSProxy. However, this made no difference and an alternative client tool was sought. The possibility of port or IP number mismatching was eliminated by continuing the development on the same machine.

The querying client made use of OSQL, a tool that comes with MS SQL Server 2008. This tool, along with packet sniffers Ethereal [20] and Packetyser [21] allowed Acknowledgments Insert acknowledgment, if any for the development of the SQL extracting method. This was done by analysis of the TDS protocol and lead to the extraction of the query in the query packet sent to the database after the login challenge.

With TDSProxy now able to capture the query sent in the TDS query packet, a vulnerable ASP application was developed. The ASP page was hosted on a remote machine and connection to the database came through the proxy server. This application allows the user to enter SQL-Injection text into the input parameters and manipulate the database.

The next step involved creating the filter which made use of powerful regular expressions. The filter uses SQL-Injection signatures which are made up of a black list, white list, gray list and pattern matching list. The filter is able to report whether the SQL query text matches any of the given signatures. At all stages of development, there is extensive logging of the queries captured. This helps with the debugging. SQL Injection attacks are logged along with the signature that caught the attack. With the aid of the log files generated by TDS Proxy and the database log files, the DBA can ascertain which database is being attacked. The DBA can also discover which web server or web page the attacker is using. The value of this is that the security holes can be patched and the database protected from further attacks.

Alerts are sent via UDP to the database administrator with the SQL-Injection query, the name of the machine hosting the web application and a timestamp. This will allow the DBA to block further injection attacks from a particular user by checking the database log file which should contain the IP address of the person who sent the query at that time. The filter method made use of black, white, gray and pattern matching signatures. When filtering was turned off, the average processing time of TDSProxy was reduced from 0.256845 milliseconds to 0.002469 milliseconds. This was for a set of 5000 queries of varied length and structure. The total signature set is 190.

Timing the latency of TDSProxy was done by subtracting the time that the database spends processing the query from the roundtrip time for a client query and response. The roundtrip time was calculated as the query enters and leaves the proxy sever on the client side only. The database processing time was calculated by timing the query and response time on the database side.

The time taken to process queries seems to be negligible given the default MS SQL Server 2008 login timeout time is 4 seconds and the default query timeout time is 0 seconds.

7. Future Work

The order of filtering may have a performance impact too. This will be investigated by changing the order that filter uses the signatures. Timing individual query execution time from a webpage will provide useful information on the impact of the TDSProxy on web interface usage. The project could be extended to handle other databases such as MySQL, Oracle and Postgres as well as other operating systems. A further extension of the project could involve an investigation into the performance impact of the proxy server on data transfer.

8. Conclusion

SQL-Injection is a relatively simple technique and on the surface protecting against it should be fairly simple. Auditing all of the source code and protecting dynamic input is not trivial, neither is reducing the permissions of all application users in the database itself. Given the research done in the area of using other methods of prevention [13] and the fact that there is a finite set of words in the SQL, it is possible to develop a filter to prevent SQL-Injection.

Checking through log files, making sure that code is perfectly secure and relying on the least privileges

principle does not seem sufficient. Detecting SQL is not as useful as preventing it. It is difficult to detect attacks and again, an audit of log is required. The use of packet sniffers does not allow for the prevention of damage as the packets collected do not allow for the removal of malicious SQL query statements. Is it viable to develop auditing software that will require a large amount of resources for computation and storage?

TDSProxy is a feasible solution to preventing SQL-Injection. The filtering process seems to provide a negligible overhead. This part of the project does however require further investigation.

Acknowledgment: This work has been done under the auspices of the Department of Mathematics, Faculty of Science, Suez University, Egypt.

References

[1] P. Finnigan, ”Detecting SQL Injection in Oracle.” http://securityfocus.com/infocus/1714.
[2] I. Ristic, ”ModSecurity for Java.” http://www.modsecurity.org/projec ts/modsecurity/java/.
[3] A. Petukhov and D. Kozlov, ”Detecting Security Vulnerabilities in Web Applications Using Dynamic Analysis with Penetration Testing, ” Proceedings of Application Security Conference, Ghent, Belgium, 19-22 May, 2008.
[4] K. Ahmad, J. Shekhar, and K.P. Yadav, ”A Potential Solution to Mitigate SQL Injection Attack,” VSRD Technical & Non-Technical Journal, 145-152, Vol. I (2), 2010.
[5] B. Indrani and E. Ramaraj, ” An Approach to Detect and Prevent SQL Injection Attacks in Database Using Web Service,” IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1, January 2011.
[6] P. Ramasamy and S. Abburu,”SQL Injection Attack Detected and Prevention,” International Journal of Engineering Science and Technology (IJEST), Vol. 4 No.04 April 2012.
[7] S. Manmadhan and Manesh T, ”A Method of Detecting SQL Injection Attack to Secure web Applications,” International Journal of Distributed and Parallel Systems (IJDPS) Vol.3, No.6, November 2012.
[8] L. Kishori and K. Sunil,” Detection And Prevention of SQL-Injection Attacks of Web Application Using Comparing Length of SQL Query,” ISSN (Print): 2278-5140, Volume-1, Issue — February, 2012.
[9] C. Anley, ”Advanced SQL injection.” http://www.nextgenss.com/papers/advanced-sql-injection.pdf.
[10] P. Finnigan, ”SQL Injection and Oracle, Part.” http://www.securityfocus.com/infocus/1644.
[11] O. Maor, and A. Shulman, ”Blind SQL Injection.” http://www.imperva.com/application defense center/white papers/blind SQL server injection.html.
[12] C. Cerrudo, ”Manipulating Microsoft SQL Server Using SQL Injection.”http://www.appsecinc.com/presentations/Manipulating SQL Server Using SQLInjection.pdf.
[13] A. Keromytis, and V. Prevelakis, ”Countering code-injection attacks with instruction-set randomization in Proceedings of the 10th ACM,” Conference on Computer and Communication Security Washington D.C., pp. 272-280.
[14] C. Hotchkies ”Blind SQL Injection Automation Techniques.” http://www.blackhat.com/html/bh-media-archives/bh-archives-2004.html#USA-2004.
[15] Microsoft,”Secure Multi-tier Deployment.” http://www.microsoft.com/technet/prodtechnol/SQL/2000/maintain/sp3sec03.mspx.
[16] Beyond Security Ltd,”SQL Injection Walkthrough.” http://www.securiteam.com/securityreviews/5DP0N1P76E.html.
[17] K. Spett,”SQL Injection Are Your Web Applications Vulnerable?.”http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf.
[18] J. Grossman,”The Challenges of Automated Web Application Scanning.” http://www.blackhat.com/html/bh-media-archives/bh-archives-2004.html#Windows-2004.
[19] Vulnwatch, ”netcat 1.11 for Windows is released.” http://www.vulnwatch.org/netcat/.
[20] Ethereal,”Ethereal.”http://www.ethereal.com/download.html.
[21] Network Chemistry,”Packetyzer - Packet Analyzer for Windows. ”http://www.networkchemistry.com/products/packetyzer/.

Yasser Fouad born in Egypt in 1972. He received his B.Sc. and M.Sc degree in Computer Science from the Faculty of Science, Suez Canal University, Ismailia, Egypt in 1994 and 2003, respectively. He received a Doctoral degree in Wireless Networks in 2009. He is a full lecturer at the Faculty of Science, Suez University, Suez, Egypt.

Khaled Elshazly born in Egypt in 1969 received the B.S. and M.S. degrees in computer science from Suez Canal University in 1991 and 2007, respectively. During 1999-2013, worked in Suez institute for management information systems. His research focuses on security network.