Centralized Dynamic Protection against SQL Injection Attacks in Web Applications

Ramzi Esmail Salah, Ammar Zahary
The Arab Academy for Banking and Financial Sciences,
Sana'a &University of Science and Technology
Sana'a, Yemen
ramzey6@gmail.com, a.zahary@ust.edu

ABSTRACT: *Structured Query Language (SQL) injection is an attack method used by hackers to retrieve, manipulate, fabricate or delete information in organizations' relational databases through Web applications. Construction of secure software is not easy task, given the complexities that may be faced. SQL injection is increasingly exploiting the weaknesses of software year after year around the world. Security relevant issues in this area had not been properly addressed in relevant literatures during the development cycle of software. This paper conducts an approach called Centralized Dynamic Protection against SQL Injection Attacks in Web Applications (CDPIA) that creates a data type for checking system to prevent data type mismatch in dynamically generated SQL queries. To strengthen the approach, CDPIA utilizes encryption technique using Rivest, Shamir and Adleman (RSA) algorithm. The paper also discusses and presents most common Web application vulnerabilities with possible attack scenarios. An implementation of the system is described by using an MS SQL written in Microsoft Visual Studio with C#. The presented approach has been tested and verified using both manual and automated method. Results show that the implemented approach can handle most common SQL injections and data type mismatches.*

Keywords: SQL Injection Attack (SQLIA), Web application vulnerabilities, Centralized Dynamic Protection, RSA Encryption Algorithm

Received: 3 March 2010, Revised 29 March 2010, Accepted 4 April 2010

©2010 DLINE. All rights reserved

1. Introduction

Internet today has become an essential part of our daily life, because of the rapid development and great services it provides, such as: shopping on the Internet, making reservations, and bills payment. With the advent of Business to Business (B2B) and Business to Customer (B2C), there is a necessity to exchange information in a safe and accurate way. Most web applications contain vulnerabilities that allow attackers to launch attacks and exploitation. Mostly in the Arab world, vulnerabilities are used in a very large because of the delayed entry of the Internet to Arab countries, lack of use among a large segment of people, and lack of extensive experience in programming. These reasons led to the emergence of these gaps which are very large in the Arabic websites compared to English websites [1]. As a result of the attacks, integrity, confidentiality and availability of information are lost. This information can be called unauthorized information which can be read or copied by unauthorized users from loss of confidentiality. Confidential information such as credit card numbers and bank records, medical records, and social security must be stored correctly so that it could not be detected [2].

Although SQL injection was discovered for a long time, modern applications such as Language-Integrated Query (LINQ) usually deal with this problem. SQL injection has been a significant risk for traditional SQL queries formed by concatenating user input. LINQ to SQL avoids such injection by using SQL parameter in queries. User input is turned into parameter values. This technique prevents malicious commands from being used by customer input. However, SQL injection is a very serious application level attack on Web applications and one of the greatest threats to Web applications [3]. Different levels of protection from one location to another are based on different levels of attack and exploiting vulnerabilities in Web applications which vary from one person to another. Restrictions on protocols (such as HTTP) are often apply firewalls and anti-virus server applications in order to prevent penetration Web sites, however protection programs and firewalls do not prevent communication over port 80. This fact can be justified because of some questions need answers. Which the provision of Internet services is required by the site, which gap is exploited by the attacker, can the access be done to things that are not authorized, and finally can firewalls, antivirus or Intrusion Detection Systems (IDSs) stop such attack [3].

The information is a wealth of basic and important resources come from the Web. As it is the most important business assets and therefore must achieve a high level of security in order to preserve information and competitive advantage, SQL Injection Attack (SQLIA) is one of the top threats that are commonly used at the level of the Web. In addition, SQLIA is one of the most serious vulnerability types that are easy to discover and apply to the infected sites to come with dire consequences, leading to the destruction of the data, manipulate, and steal the rights and money of the users [4].

There are many reasons of why SQLIAs are commonly used. For example: theft of confidential data, financial fraud, espionage and terrorism, electronic warfare, destruction of data, just a hobby and fun. Therefore, the techniques used in SQLIAs have become more common and sophisticated. Thus, there is an urgent need for the solutions of this problem, especially in legacy systems. Detection or prevention of SQLIAs is a topic of the recent research in the industry and academic sectors. However, none of them is complete or accurate enough to guarantee an absolute level of security in Web applications [5].

This paper tries to prevent SQL injection, especially in legacy systems by conducting an approach called Centralized Dynamic Protection against SQL Injection Attacks in Web Applications (CDPIA). The approach CDPIA has developed a data type checking system that prevents data type mismatch in dynamically generated SQL queries. To strengthen the approach, CDPIA also utilizes encryption technique using Rivest, Shamir and Adleman (RSA) algorithm. The approach has been implemented and evaluated in this paper by developing a Web system using an MS SQL written in Microsoft Visual Studio with C#. The developed system has been tested and evaluated using both manual and automated methods. Results show that the implemented system can handle most common SQL injections and data type mismatches.

2. Related Work

There are so many scientific researches that are relevant to SQLIAs detection and prevention on a large scale area. The related researches have been classified based on the type of data analyzed or modified by the proposed techniques to make it easier to deal with them such as: runtime HTTP requests, design-time Web application source code, and runtime dynamically generated SQL statements. To detect SQLIAs, some approaches use only one type of data while the other use two types. For example, our approach analyzes HTTP requests and SQL statements. In this section, we discuss related work according to the categorization described above.

2.1 Runtime Filtering of HTTP Requests

Security gateway [12] is a filtering proxy that only allowed inputting the correct order to reach the Web application. If HTTP requests are compliant with the input validation rules, then allow to reach the protected Web applications. Similar to commercial Web application firewalls, this approach is easy to operate and deploy without any modifications to the application Web source code. Security gateway needs developers to provide correct validation rules which are specific to their application. Also, similar to the defensive programming practices, this process requires intimate knowledge of the Web application in question and as a result, it is prone to false positives and false negatives. In addition, filtering HTTP requests is performed only without checking text input. Any modification of an existing deployment or Website application of a new one requires modification to the input validation rules leading to an increase in the administrative and change management overheads. By comparison, our approach does not require deployment of interception modules and does not need developer involvement, only it needs adding centralized protection.

2.2 Web Application Source Code Analysis and Hardening

Approaches proposed by Jovanovic et al. [6], Xie et al. [15], and Livshits et al. [8], use information-flow-based source code analysis techniques to detect SQLIAs vulnerabilities in Web applications. Once detected, these vulnerabilities can be fixed by the developers.

These approaches of vulnerability detection employ static analysis of applications. They have the advantages of no runtime overhead and they have the ability to detect errors before deployment. But they need access to the application source code, this is access is sometimes unrealistic. May also lead to problems with legacy systems, and the analysis have to be repeated each time the application is modified. Repeated analysis increases the overhead of change management. By comparison, our approach does not require access or update to the source code, only add centralized protection.

2.3 Runtime Analysis of SQL Statements for Anomalies

Valuer et al. [14] propose a technique for SQLIAs detection based on artificial intelligence using machine learning methods. Their anomaly-based system learns profiles of the normal database access performed by Web-based applications using a many of different models. These models allow the detection of unknown attacks with limited overhead. After learning normal profiles in a training phase, the system uses deviation from these profiles to detect potential attacks. In [14], authors have shown that their system is effective in detecting SQLIAs. However, the fundamental limitation of this and other approaches based on machine learning techniques is that their effectiveness depends on the quality of training data used. Training data acquisition is an expensive process and its quality cannot be guaranteed. Non-perfect training data causes such techniques to produce false positives and false negatives. By comparison, our approach does not rely on the ability of the application developers or owners to acquire a qualified perfect data set, which has all possible versions of legitimate SQL statements and yet has no SQLIAs.

2.4 Static Analysis Paired With Runtime Analysis Of SQL Statements

CANDID [2], SQLGuard [4], AMNESIA [5] and SQLCheck [13] adopt this method to identify the intended structures of SQL statements by analyzing the source code of Web applications at development time and checking at runtime whether dynamically generated SQL statements conform to those structures. SQLrand [3] modifies SQL statements in the source code by appending a randomized integer to every SQL keyword during design-time. An intermediate proxy intercepts SQL statements at runtime and removes the inserted integers before submitting the statements to the backend database. Therefore, any normal SQL code injected by attackers will be interpreted as an invalid expression. These approaches are very effective and claiming 100% accuracy. The key is not known to the attacker, so the code injected by attacker is treated as undefined keywords and expressions which cause runtime exceptions and query is not sent to database. The disadvantage of this system is need access to the application source code for the purpose of analysis and modification, which is their main limitation, And May also lead to problems with legacy systems, and complex configuration and the security of the key. If the key is exposed, attacker can formulate queries for successful attack. By comparison, our approach does not require update SQL statement to the source code, only add centralized protection.

2.5 Runtime Analysis of HTTP Requests and SQL Statements

Nyguyen-Tuong et al. [10] and Pietraszek et al. [11] approaches employing dynamic taint analysis. Taint information refers to data that come from un-sanitized or un-validated sources, such as HTTP requests. The two approaches modify the PHP interpreter to mark tainted data as it enters the application and flows around. Before any database access function (e.g., MySQL query()) is dispatched, the corresponding SQL statement string is checked by the modified PHP interpreter. If tainted data has been used to create SQL keywords and/or operators in the query, the call is rejected.

By comparison, our approach uses HTTP requests and SQL statements. We do not need to search the depths of application source codes. The application might include thousands of lines in the code. Our approach is resistant to evasion techniques. Their limitations are that they require modifications to the PHP runtime environment, which may not be viable for other runtime environments such as Java or ASP.NET, and need all database access functions to be identified in advance. Our approach runtime is ASP.NET environment and the approach can easily be adapted for Web applications written in other languages.

3. CDPIA Approach

In this section, we provide a full description of our proposed methodology Centralized Dynamic Protection against SQL Injection Attacks (CDPIA) to detect and prevent SQLIAs. We offer the full implementation of the prevention of SQL injection, and we have implemented the program with Microsoft visual studio 2008 with C# and Microsoft SQL server 2008, where the steps to prevent SQL injection tools discussed in details. We also offer a central solution step by step procedure to achieve this. We also encrypt important data with RSA algorithm and then analyze the details of this step. Then, we discuss the advantages and limitations of our proposal.

The proposed method on the basis of which include a logical start easy and then start more difficult, and is based on the central role of protection from attacks by hacker. The method is based on the basis of examination of all inputs from the user, and the presumption of the existence of malicious characters in any text input that is important. Therefore, it is examined first by HTML directly. This method is very easy and does not need to code. It is also useful in the quick fix that does not accept the introduction of any symbolic characters. However, it is not effective in solving all problems because some browsers do not support this feature at all, and there are also fields requires the introduction of such symbolic characters for one reason or another. It must be allowed to pass and hide any error to appear in the event of any error message. The error is sent directly to a supportive environment and the error page is transformed to the main page of the site.

In case of failure of the first method, or the possibility of overcoming them, the second method will be used automatically. This method is based on the examination of any input from the user that is being tested by the central examination. If the text entry does not contain any threat, it is passed to the database. However, if the entered text contains one or more threats, it is divided into two parts; Part I: a common mistake that is used by the general users. Usually users do not mean any threat. Many of them use such as the use of {'} in the course of the text. In this case the code is replaced by a different code, without the user's knowledge, and the input is passed to the database. Part II: use local file indicates the presence of malicious input and this often contains two or three symbols together represent a threat which is prevented from passing inputs. It also sends an alert message containing the IP address and browser type. This information is stored in the database and other data will be barred if the attacker tried more than three times. In the final stage of inputs test, an important input is encrypted, and stored in an encrypted database.

The method checks HTTP request by central function by sending each request for examination. If no threat is found, it allows the request. However, if any threat is found, it does not allow the request, then it moves to another page and saves data about this hacker, and finally it issues an alert message.

3.1General Model Of CDPIA Approach

Our methodology, CDPIA, is conducted to detect and prevent SQLIAs. We proposed Centralized solution through the examination of all input from the user, whether the text input throw step five, input validation, hiding the error messages, function security, and HTTP Request validation, and we evaluate our security system for different tests, and it worked correctly to prevent SQL injection attack. All of this is clear from the following in Figure 1, and will explain all in detail.

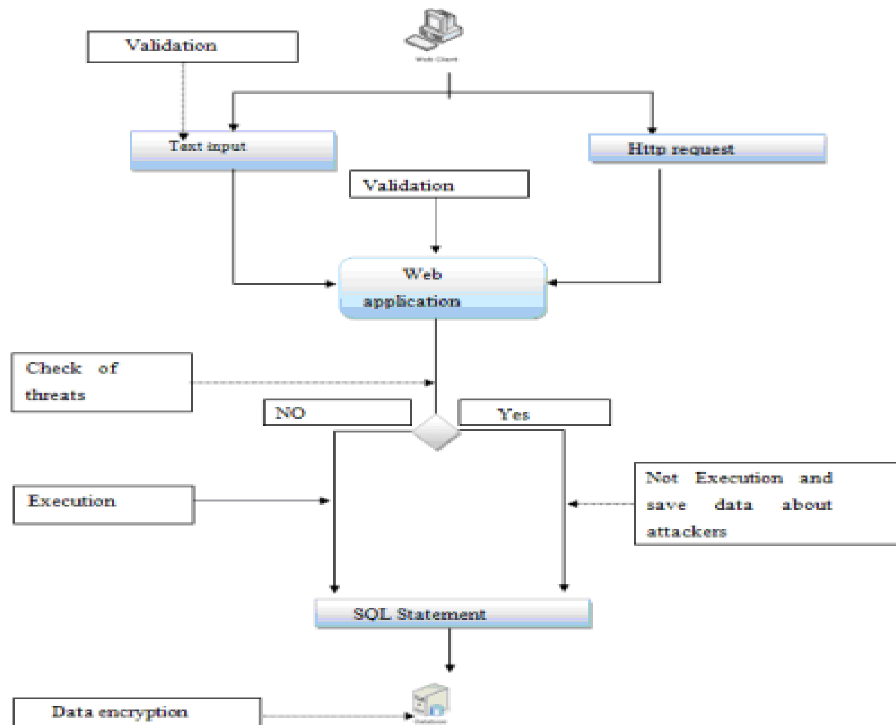

Figure 1. CDPIA approach: General Model

3.2 Input Validation

Every passed string parameter should be validated as shown in Figure 2. Many Web applications use hidden fields and other techniques, which also must be validated. If a bind variable is not being used, special database characters must be removed or escaped. For Microsoft SQL (MSSQL) databases, the character at issue is a single quote. The simplest method is to escape all single quotes. The use of bind variables and escaping of single quotes should not be done for the same string. A bind variable will store the exact input string in the database and escaping any single quotes which leads to store double quotes in the database.

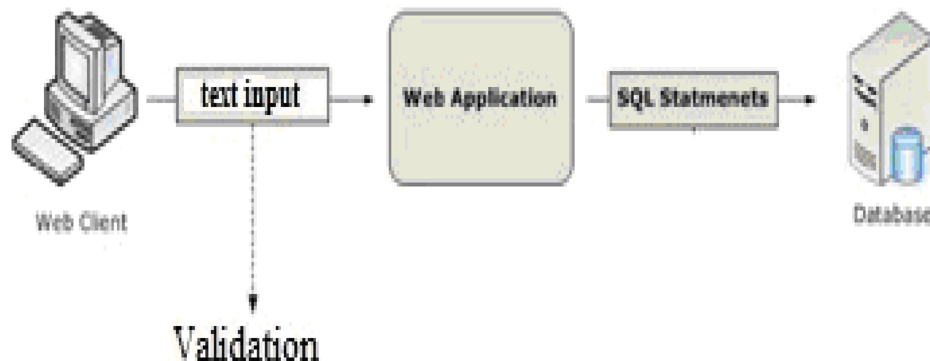

Figure 2. Input validation

3.3 Hiding Error Messages

A general error page can be set so that no one can get the actual database information. However, a custom error page can be created by which in the error page, we can implement an automatic error reporting system which triggers an email to the programmer with a detailed error report while the error occurs. As usual, all parameters are tested individually, with the rest of the request being valid. This is extremely important in this case as this process must neutralize any possible cause of error other than the injection itself. The result of this process is usually a long list of suspicious parameters. Some of these parameters may indeed be vulnerable to SQL injection and may be exploited. The other parameters had errors that are unrelated to SQL, thus they can be discarded. The next step for the attacker is therefore identifying the pick of the litter, which, in our case are those that are indeed vulnerable to SQL injection.

3.4 Function Security

It is important to block hackers from manipulating the URL query string and text boxes to block their inputs. However, how do you determine who they are, what they will input and whether or not it is a safe process? Unfortunately, you could not know. Thus, you must assume that all user inputs could be potentially dangerous. A common saying in the programming world is that “ALL INPUTS ARE EVIL”. Thus, it must be treated carefully.

Everything comes from everybody should be checked every time to ensure dangerous code does not slip in. This is accomplished by checking all inputs that are submitted via a query string or form, and then rejecting or removing unsafe characters before it reaches the database. If this sounds like a lot of trouble, you are right. But, it is the price we pay to protect our Websites and databases from the wrath of hackers. It is your responsibility as the Web master to ensure that only clean and safe input is allowed to enter your database. We propose many ways to block this type of injection. The recommended process is to use stored procedures with parameterized SQL because they are safe and length specified. However, for a large existing application which never used any parameterized query, it will be a time consuming task to convert every dynamic query to parameterized query. A quicker solution is to build a central monitoring system, which validates input variables from all forms. Figure 3 illustrates the concept of function security.

There are many ways to collect data in a form. We normally use post variable, query string, and cookie variables to pass information from one page to another in ASP. To protect our application from SQL injection attacks, we need to validate inputs by checking the input type, length, format, range, etc. Also, we have to validate that no harmful SQL keyword is used as input data such as drop, declare, cementation (’–’), execute, varchar, char, etc. Thus, first of all, we have to create a black list by which we can detect harmful execution.

Validation can be done in both client side and server side. Do not rely on client side validation, since it can be easily bypassed by disabling JavaScript. Thus, server side validation is necessary. Client side validation can be used to improve the user experience and server performance by reducing round trips. We can consider the following functions presented in the next sub section to validate input string.

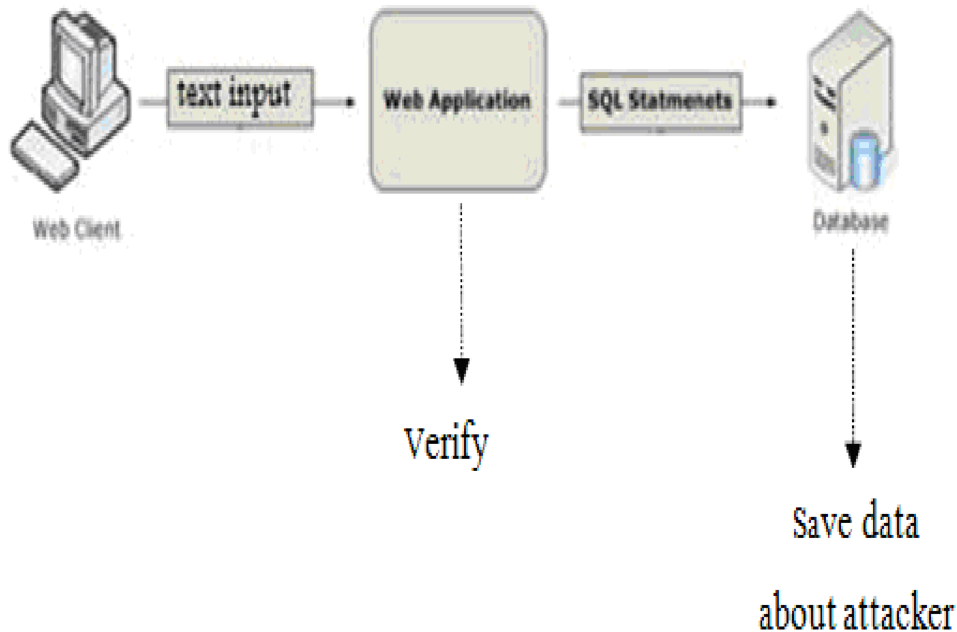

Figure 3. Function security

3.5 Safe And Bad Characters

Part one of the validation process is to reject all inputs unless the input contains safe characters. This is the strictest and most effective form of input validation. It only allows input that is known to be good. Essentially, letters and numbers can be trusted. Special characters are the real culprits which give hackers their power and should be avoided. This extreme measurement may not be feasible to all types of inputs, but it tries to restrict as many special characters as possible.

Part two of the validation process is to reject all inputs if the input contains bad strings. After you have collected good inputs using this method, you should check again for inputs that are known as bad. Dangerous things could happen if the good character function allowed an apostrophe and hyphen, or other letter combinations like SCRIPT, SELECT, UPDATE, DELETE, etc. That is why the bad string function should be used in conjunction with the good character function.

Another method that can be used in conjunction with the above two functions is "filter characters" however, it is considered very weak when used alone, because it sanitizes the input by filtering or escaping.

4. Implementing CDPIA with RSA Algorithm

It is quite difficult to understand the commands from the down inputs. However, after decoding the HEX code to ASCII string, it becomes easy to understand the commands of a code. We use a technology that utilizes RSA algorithm in order to increase the protection of a database in the event of any threat or exposure to a person. Also, the algorithm increases the confidentiality and safety is inherited. We apply this algorithm to the encryption process of the database if it is exposed to the rules of data in any way that the data could not be useful in the future at all, because the encryption algorithm is impossible to escape. Encryption with RSA has been applied to our approach CDPIA as shown in Figure 4.

We can screen all incoming query-string, form and cookie values by running code during the BeginRequest event. This type of code can run on every request when implemented in an HttpModule. The sample code below defines an HttpModule in the App Code directory, and then registers the module in Web.config so that it runs on every request. The sample code will check incoming data and automatically redirect to a page called "Error.aspx" if suspicious character sequences are found as

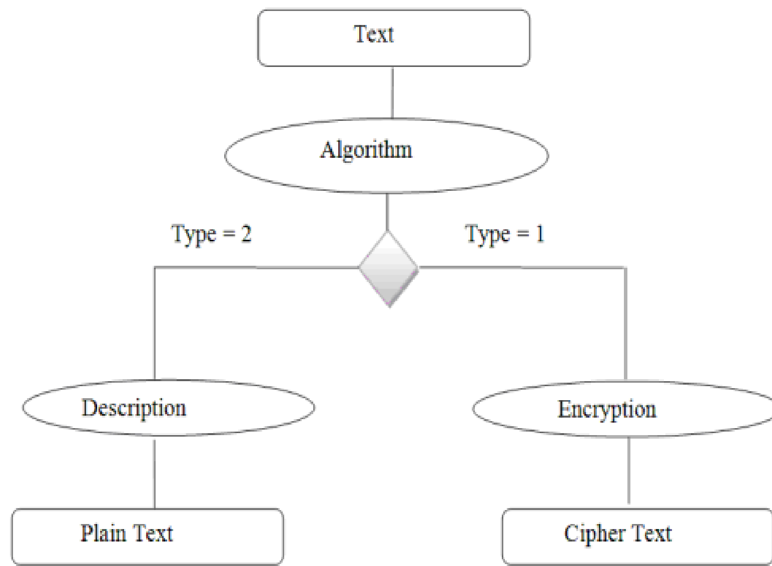

Figure 4. Encryption with RSA algorithm

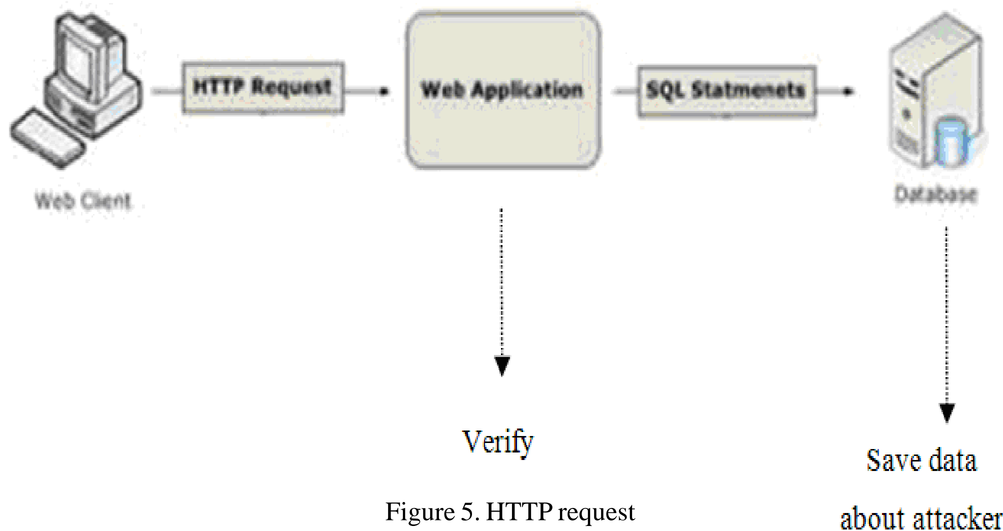

Figure 5. HTTP request

illustrated by HTTP request in Figure 5.

5.CDPIA Evaluation

In this section, the evaluation of our approach CDPIA will be presented. The software of CDPIA has been evaluated for SQL injection vulnerabilities in two ways, static and dynamic evaluations. Static evaluation has been done using list of malicious inputs provided by some of the typical commands used by hackers to run SQL injection attacks on Web applications, while dynamic evaluation has been done using the automated service-based programs inspected the site.

5.1 Static Evaluation

Static evaluation is typically done by performing source code analysis. This method creates a control flow graph of information that is processed by a server page. The graph consists of input and output nodes. An input node can be define as a statement

that processes input data from a form, reads the value of a query string, a database field, a cookie, or data from a file.

Output nodes are associated with statements that write to database fields, a file, a cookie, or output in the page. The server page is potentially vulnerable if a path in the control flow graph exists so that it connects an input to an output node. However, it is possible that some data is sent from one server page to another page. In this case, the Web application might not be vulnerable to a certain type of attack if only one of the individual server pages has potential security problems. For example, a page may read input and store it in a database field. The result of the static analysis says that this page has a potential vulnerability. However, another page that reads data from this field may encode everything in the output of the page and therefore, the Web application as a whole is not vulnerable.

We have implemented all strings malignant and apply them as one unit which has to make sure that all of them can never run SQL injection attacks on Web applications during the examination of the central control that we have done. The testing process is based on the basic inputs that cannot be correct without SQL injection, which is standard, so that no one can process subversion of the SQL to any other injection.

5.2 Input Validation

Input validation refers to the process of validating all inputs of an application before using them. Input validation is absolutely critical to application security, and most application risks involve tainted input at some level. Several attacks can be run against a Web application that insert malformed data (often, too much at once) which can confuse, crash, or make the Web application divulge too much information to the attacker. Our approach prevents hackers from using any symbolic characters, except characters a-z, or numbers 0-9. The validation is directly shown without need to examine the code, but this is accomplished through the html directly. Thus, this scan is fast, easy and very useful in certain places however, in some places you cannot use the last method if you have a field that requires the existence of such symbols, and then this person will be able to add whatever he/she wants. Finally, the user can make sure about the code of the function. The examination of inputs has been presented in this paper to ensure the integrity of the text through the HTML before it is processed. As shown in Figure 6, when entering anything unwanted to the application, it will be arrested immediately.

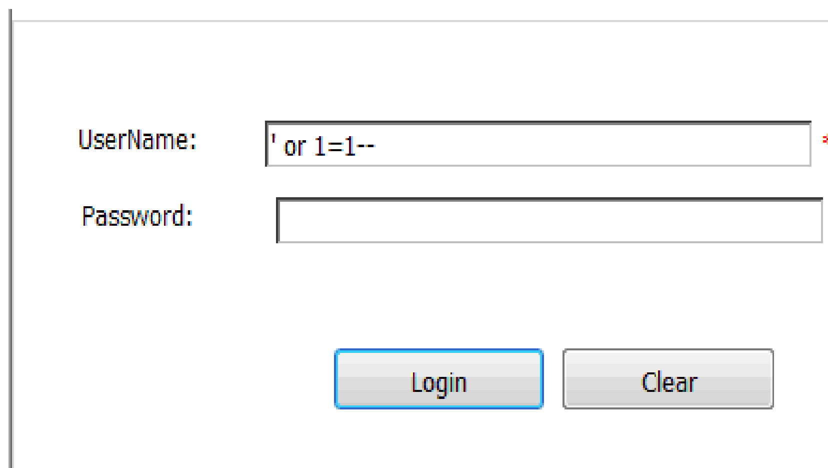

Figure 6. How input validation works

5.3 Error Messages

An ASP.NET application must enable custom error pages in order to prevent attackers from mining information from the framework's built-in error responses. The custom error page should be configured instead of the framework default page. In the event of an application exception, generic error messages should be returned to the client. The default error page provides detailed information about the error that has occurred. Attackers can leverage the additional information provided by a default error page to mount attacks targeted on the framework (database), or other resources used by the application. Our approach also tries to prevent the emergence of any error message that would indicate anything to the attacker or leave a guide to hacker to know anything. In case of these indications has happened the user will be transferred directly to the homepage by default, and a generated error message will be sent automatically to the support of the application.

5.4 Security Function

This process concerns the examination of any entrance to the database through any user. In the security function process, we assume that all inputs are incorrect and therefore all inputs will be examined anywhere in the Website and only the correct input which do not cause any problems will be allowed to pass to the database. In our approach, this operation has been conducted in three stages. First stage is based on the rule states that "Do not allow any incorrect input to cross into the database".

Second stage is to prevent any input that contains some types of risk all together including the risk of bad characters.

Third stage is to delete common errors that may exist without a direct order like {'}, and then to allow the passage of the rest. By applying these stages to all inputs that contain different SQL injections, all injections are successfully prevented. Minor mistakes are allowed to pass for testing, these errors are deleted, and the rest of characters are allowed.

The implementation of this waiver has been done in the case of some letters only which have be deleted first before allowing them to be saved to the database.

5.5 Encryption With RSA Algorithm

This property has been added in order to maintain the confidentiality of data in case of exposure. The adoption of encryption was for all task fields that actually need it. Encryption will help in two cases: first case is that no SQL injection can pass through encryption, and the second is the case of exposure which cannot be able to take advantage of using this data because of the adoption of RSA algorithm encryption which keeps the data as safe as possible and does not allow the data to be dismantled if the private key is not known.

For example, we have added two users, the result was as follows: Encryption text: Ramzey to:
6B7Y9B6H8W5G2C5A2G9O2W4B9D9F0D2X0X7O6J2XL3N1Q6P3D9S1V2P0N0R7M4A9H7Z5C9E0D5R5N1A8C
5K4G8X2R7V6Q 0U3T9B 7X2 N1O 7L9D 1D0V 3Y2U5V6U 0M7 K0P5M 9J7U4D4 P9L6I8A3P9D1P8L8S2 S8R0K2W4P8V0U6B
1H5R7N3N5Q6V3O2M7L2M0H4G9O6R1K9R4B1Q3X8B7H1G8N0Y4D3D7U4K6S3R1T3H8Y5P0B6U3L7D8O4J7M9V6V

Encryption text: Ammar to:
4O8Y3G5W5S8A5Q7C9O3L3Q9D1Z3J1O2P7X8A4F7N 7V6A8W9L2T3E3Y8S0P7A7J7L0O3V9C 3M2E2P1I5K9L 6Q9Q3I9C6L6
T7D8Z8Q9E0G6S9L3R2 T0Y4Q0U4W 2F6M8Q1O8L1O6S7D2V1U3W3K1X3H7A1X1G5S 2S8J0V5Q1C7W 1F5W7F5D9V5Y0
Y4U1J4E5S0U8Y6Y1N7L2Y1F1P2Z6R0L0O9S8K8K5Q4Y6S2G1W2O1U1D5Z7M8U3A5I2X3E6B3A9D

And, we gave the two users the same password: R123. For the first user, the text is encrypted to:
5Y8N9F3K8U4P0F8B3O0Z8I2 L3I4V9I8U8 M1F1Y6M4V7B0J4W5B9C7J0P5W9L1T0 N0V9H9U0 S4M6C3U2W6X6Q3J1R2Y
4H8Y1R6T7D3U5S2Y3V8O7B5B7A4V6Y8R5X3A4V8F0P6W6A3O7U1S4W3B3L5M0Y1V4K7M0U1S8Z9G0G5G
0Q6A8N5F1E4 C4 F7S4D3Z3J8D7Q8J4V4Z6L6S4M5L5Z4D7H1K1Y6P3Z1L8D9T5P5Q0O9N9M0F6D5E2M9Q3F4A9Q

For the second user, the text is encrypted to:
5I6Q1R8Z7F3C2E3N9E8B5J5U5A8W8F0C7E1G1Y8A9E0U5H8B6O4C6N5L9B7R9E2X3C6X2M7S4B3L0H6N4D3X5U5
R4W3T2D6C2K8H1W4S2L9V6M6T4L6F8F4C7L4F4R3P0Q8X3G9N2H1L1W3O9D2M0I8S5T6X6Y5A2N2V4K1O2W
0M5U9I1K8V6G1M6A8L5Y7H7Z2V0X5Y2W0E4Z8P7E8Q5L8R1K7L7Z3R4O7S5R7E4C8V8N6R8B2R9R5X3P1E8G3E

There is a difference between the encryption of the first and the second user for the same word. Cipher text using RSA algorithm is shown by Figure 7.

5.6 Saving the Attacker Data

Validate all parts of the HTTP request, check all inputs of HTTP request, and then prevent characters that may cause SQL injection. This process has been applied to both examples of existing applications, and it was successful in preventing all expected intrusions.

In this process, we save the attacker information. If the attacker tried more than three times, he/she would be prevented from entering the site for a specified period. This process is useful to know some statistics about the attacker at anytime, and anywhere, and also to know how words are used. Figure 8 shows the process of saving the information about the attacker.

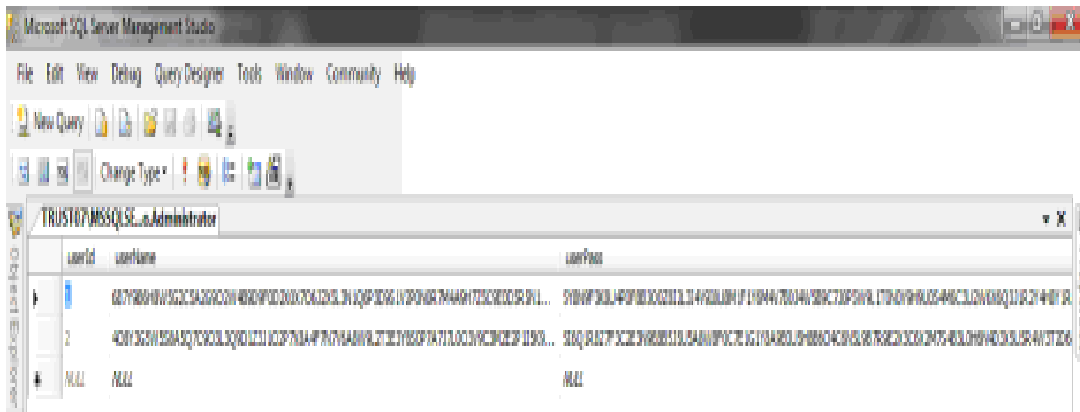

Figure 7. Cipher text using RSA algorithm

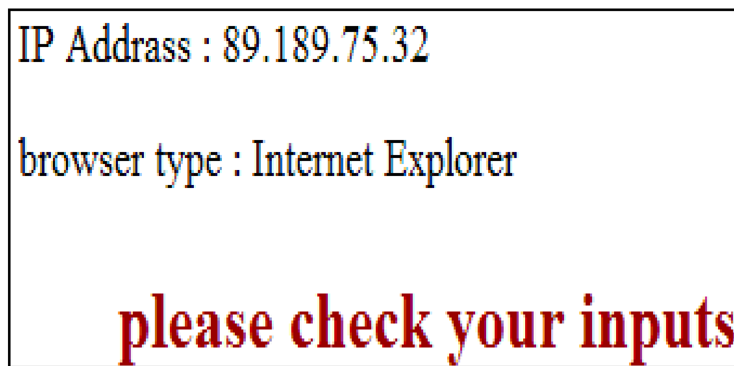

Figure 8. Saving the information about the attacker

5.7 Dynamic Evaluation

In dynamic evaluation, known attacks are executed against Web applications either using a database with generated attacks for a specific Web application or a database that contains generic attacks. Then, the users implement dynamic testing as a second stage of their Web application assessment. More precisely, server pages that are potentially vulnerable according to a previous static analysis step are tested again in a dynamic test.

With specific attacks for the potential vulnerability, the crawler is described in a black box testing with a generic database. It analyzes the generated pages of the Web application and then chooses attacks to perform. This method tests Web applications without requiring user interaction and then interprets the response of the Web application to the chosen attack.

Software analysis is a powerful method that is used to detect possible vulnerabilities. However, the source code is necessary to static analysis to be effective. Dynamic tests allow us to test code without the need of source code, however it can only test known vulnerabilities. Black box testing approaches usually analyze the Web page and may not find every possible input that can be used to perform attacks.

Website security is possibly today's most overlooked aspect of securing the enterprise and should be a priority in any organization. Hackers are concentrating their efforts on Web-based applications such as shopping carts, forms, login pages, dynamic content, etc. Web applications are accessible 24 hours a day, 7 days a week and control valuable data since they often have direct access to backend data such as customer databases. Firewalls, SSL and locked-down servers are futile against Web application hacking.

Any defense at network security level will provide no protection against Web application attacks since they are launched on

port 80, which has to remain open. In addition, Web applications are often tailor-made, thus they are usually less tested than off the-shelf software packages, and therefore they are more likely to have undiscovered vulnerabilities. Acunetix WVS [1] is well-known testing software that automatically checks your Web application for SQL Injection, XSS and other Web vulnerabilities. We have examined this application using a well-known global program called Acunetix, which is used by global corporations through which the application was examined and gave the report illustrated by Figure 9 in which details about scan of our Website can be seen. In our Website, no vulnerabilities have been discovered by the scanner.

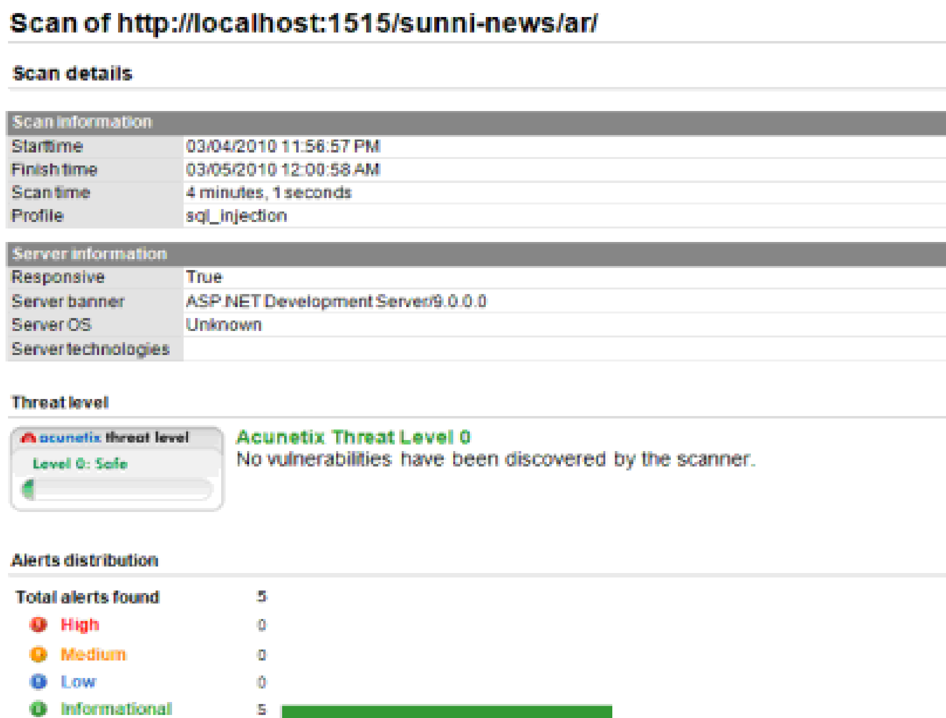

Figure 9. Scan of our Website

In the report illustrated by Figure 10, details about scan of testaspnet.acunetix.com can be seen. In Figure 10, we can see one or more high severity type vulnerabilities that have been discovered by the scanner. A malicious user can exploit these vulnerabilities and compromise the backend database and/or deface the Website application. Such scripts are possibly vulnerable to SQL injection attacks. SQL injection is a vulnerability that allows an attacker to alter backend SQL statements by manipulating the user input. An SQL injection occurs when a Web application accepts user input that is directly placed into an SQL statement and does not properly filter out dangerous characters. This is one of the most common application layer attacks that are currently being used on the Internet. Despite the fact that it is relatively easy to protect against attacks, there is a large number of Web applications vulnerable. An attacker may execute arbitrary SQL statements on the vulnerable system. This may compromise the integrity of the system’s database and/or expose sensitive information. Depending on the backend database in use, SQL injection vulnerabilities lead to varying levels of data/system access for the attacker.

It may be possible to not only manipulate existing queries, but to UNION in arbitrary data, use subselects, or append additional queries. In some cases, it may be possible to read in or write out to files, or to execute shell commands on the underlying operating system. Certain SQL Servers such as Microsoft SQL Server contain stored and extended procedures (database server functions). If an attacker can obtain access to these procedures, it may be possible to compromise the entire machine. In the reports illustrated by Figure 9 and Figure 10, details can be seen about both scans of our Website and testaspnet.acunetix.com. In our Website, no vulnerabilities have been discovered by the scanner, while in testaspnet.acunetix.com, there are one or more high-severity type vulnerabilities that have been discovered by the scanner. A malicious user can exploit these vulnerabilities and compromise the backend database and/or deface your Website. The difference in results between our Website and the Website testaspnet.acunetix.com can be justified by that our site is built using our approach CDPIA with almost no penetration, while the other Website is vulnerable because there are more than a hundred of threats that have been recorded by this Website.

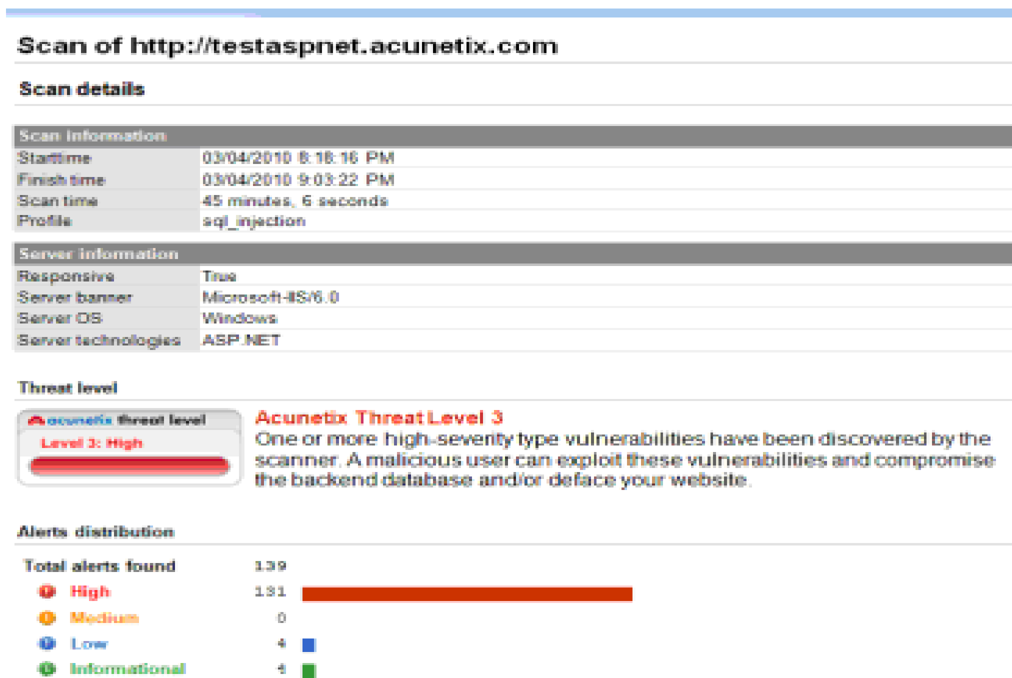

Figure 10. Scan of testaspnet.acunetix.com

6. Conclusion

In this paper, we have developed a security approach called CDPIA that uses static and runtime analysis of SQL statement to prevent SQL injection attacks of the most common Web application vulnerabilities. Our approach has been implemented using an MSSQL written in Microsoft Visual Studio with C#. The evaluation of the approach has been performed for a central line with regular, large and small examination applied to the inputs through a central scanner. RSA algorithm has been implemented also in CDPIA to encrypt the important information. Static evaluation has been applied to all strings that are presented in malignant and dynamic evaluation which has been applied using Acunetix application. By testing two Websites, our Website and the Website of testaspnet.acunetix.com., our approach CDPIA has performed well in our Website and prevented most SQL injection attacks while the other Website has recorded more than a hundred of threats. Currently, our approach CDPIA works only for .NET based Web applications, thus further work needs to be done to make it a more generalized system. Also, a researcher can apply effective centralized dynamic protection against all Web vulnerabilities and all system components so that any application can be installed on any system regardless its environment.

References

[1]WEB Application Security, by acunetix.com. http://www.acunetix.com/, 1st March 2010.

[2] Bandhakavi, Sruthi., Bisht, Prithvi., Madhusudan, P., Venkatakrishnan,V. N. (2007). CANDID: Preventing SQL injection attacks using dynamic candidate evaluations, *In:* Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS), p. 12-24, Alexandria, Virginia, USA, October.

[3] Stephen, W. Boyd ., Keromytis, Angelos D.(2004). SQLrand: Preventing SQL injection attacks, *In:* Proceedings of the Second International Conference on Applied Cryptography and Network Security (ACNS), p. 292-30.

[4] Gregory, T., Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. SQL Guard: Using parse tree validation to prevent SQL injection attacks. *In:* Proceedings of the 5th International Workshop on Software Engineering and Middleware, p. 106-113, Lisbon, Portugal, September 2005.

[5] William, G.J., Halfond and Alessandro Orso. (2005). AMNESIA: Analysis and monitoring for neutralizing SQL injection attacks. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pages 174-183, Long Beach, California, USA.

[6] Jovanovic, Nenad., Kruegel, Christopher., Kirda, Engin (2006). Pixy: A static analysis tool for detecting Web application vulnerabilities (short paper). *In:* Proceedings of the 2006 IEEE Symposium on Security and Privacy, May 2006.

[7]Kosuga, Yuji Kono, Kenji Hanaoka, Miyuki., Hishiyama, Miho., Takahama,Yu (2007). Sania: Syntactic and semantic analysis for automated testing against SQL injection. *In:* Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC), December 2007.

[8] Livshits.V., Benjamin., Lam,Monica S.(2005). Finding security vulnerabilities in Java applications with static analysis. *In:* Proceedings of the 14th USENIX Security Symposium, p. 271-286, August.

[9] MITRE (2009). Common vulnerabilities and exposures list. http://cve.mitre.org

[10] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening Web applications using precise tainting. *In*:Proceedings of the 20th IFIP International Information Security Conference, pages 296-307, Makuhari-Messe, Chiba, Japan, 2005.

[11] Pietraszek, Tadeusz., Vanden Berghe, Chris (2005). Defending against injection attacks through context-sensitive string evaluation. *In*: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, pages 124-145, 2005.

[12] Scott, David., Sharp, Richard (2002). Abstracting application-level Web security. *In*: Proceedings of the 11th International Conference on the World Wide Web, p. 396-407, Honolulu, Hawaii, USA, May.

[13] Su, Zhendong., Wassermann,Gary (2006). The essence of command injection attacks in Web applications. *In*: Proceedings of the 33rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages, p. 372-382, Charleston, South Carolina, USA, January 2006.

[14] Valeur, Fredrik., Mutz, Darren.,Vigna, Giovanni. (2005). A learningbased approach to the detection of SQL attacks. *In* Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005), p.123-140.

[15] Xie,Yichen., Aiken, Alex (2006). Static detection of security vulnerabilities in scripting languages. *In*: Proceedings of the 15th USENIX Security Symposium, p. 179-192, August.