Improving Web Application Firewalls to detect advanced SQL injection attacks

Abdelhamid Makiou, Youcef Begriche, Ahmed Serhrouchni

Improving Web Application Firewalls to Detect Advanced SQL Injection Attacks

Abdelhamid MAKIOU
Youcef BEGRICHE
Ahmed SERHROUCHNI

Telecom Paristech 46, Rue Barrault 75013 Paris France
E-mails: makiou@telecom-paristech.fr youcefbegriche@ieee.org ahmed@telecom-paristech.fr

***Abstract*—Injections flaws which include SQL injection are the most prevalent security threats affecting Web applications[1]. To mitigate these attacks, Web Application Firewalls (WAFs) apply security rules in order to both inspect HTTP data streams and detect malicious HTTP transactions. Nevertheless, attackers can bypass WAF's rules by using sophisticated SQL injection techniques. In this paper, we introduce a novel approach to dissect the HTTP traffic and inspect complex SQL injection attacks. Our model is a hybrid Injection Prevention System (HIPS) which uses both a machine learning classifier and a pattern matching inspection engine based on reduced sets of security rules. Our Web Application Firewall architecture aims to optimize detection performances by using a prediction module that excludes legitimate requests from the inspection process.**

***Index Terms*—SQL injection - Web Application Firewall - HTTP dissection - machine learning - Security rules**

I. Introduction

Structured Query Language (SQL) injection is one of the most devastating vulnerabilities that impacts DataBase Management Systems (DBMS), as it can lead to the exposure of all the sensitive information stored in an application's database [2]. In order to confront SQL injection attacks, various methodologies and techniques have been used. On one hand, Web application developers adopted safe coding and applied input validation functions. They developed filters to protect the application's entries from SQL code injections. These filters block inputs that contain SQL keywords or special characters commonly used in malicious SQL code injection. On the other hand, Web Application Firewalls protect application's database by inspecting HTTP traffic and applying a set of security rules. The language used for expressing security rules can explicitly describe a signature of an SQL injection attack, or implicitly describe the way of detecting these attacks. It can also express an anomaly score value which increases every time a malicious pattern appears in an HTTP request. If the anomaly value reaches a predefined threshold, the request will be rejected. In spite of the robustness of the above methods, attackers can bypass them by substituting malicious pattern characters or varying its format. However, a basic solution is to write a specific rule for each type of evasion technique, but it requires a high mastery of both HTTP protocol and regular expressions programming. Furthermore, pattern matching algorithms, used by security rules in order to inspect complex patterns, decrease the overall performances of the detection engine. Since pattern matching algorithms require a lot of resources, some WAFs [7] [9] are configured to only inspect POST request.

In this paper, we propose a hybrid approach to detect SQL injection attacks and their evasion techniques. Our proposal enhances both the inspection process of HTTP streams and security rules management.

Our contributions can be summarized as follows:

1) A brief survey on filters evasion techniques including code obfuscations and headers-level SQL injections.
2) A novel method to dissect and parse the HTTP protocol, that enhances security rules management, which makes the formalism of writing security rules less complex.
3) A Hybrid Injection Prevention System (HIPS) architecture, by introducing a supervised machine learning module. The classifier (machine learning module) predicts SQL injections and forwards suspicious headers to the rules-based detection engine in order to be deeply analyzed .
4) An evaluation of the effectiveness of the proposed method based on false negatives impact instead of false positives which are commonly used in Anti-spams cost evaluation.

II. Related Work

In [3][4], Kruegel and Vigna propose an anomaly-based intrusion detection system for web applications. It characterizes HTTP requests using a number of statistical characteristics derived from parameter's length, character distribution, structure, presence and order. This method focuses only on the incoming query parameters whereas it ignores the corresponding HTTP response. These results are either causing unnecessary false positives or missing certain attacks. AMNESIA [5] is an SQL injection detection and prevention system which combines static analysis and run-time monitoring. It uses a model-based approach to detect illegal queries. Nonetheless, it requires web application's source code reviewing. In SQLrand [3][6] instead of normal SQL keywords developers create queries using randomized instructions. In this approach a proxy filter intercepts queries to the database and de-randomizes the keywords. By using the randomized instruction set, attacker's injected code could not have been constructed. As it uses a secret key to modify instructions, security of the approach is dependent on attacker ability to seize the key. It requires the

978-1-4799-8099-4/14/$31.00 ©2014 IEEE

integration of a proxy for the database in the system as the same as developer training. ModSecurity WAF, proposed by Ivan Ristic [7], is an open source solution based on signature attack detection. ModSecurity is widely used and has medium performances. Though, this system is strongly related to some types of web servers and it only analyses POST queries in order to avoid performance deterioration. In addition, the rules formalism is very complex which requires a high expertise in HTTP protocol and in regular expressions coding. IronBee [8], a new project similar to ModSecurity, aims to improve detection performance and facilitate the expression of security rules by introducing the LUA scripting language. An other recent open source project NAXSI [9] uses a heuristic approach for the detection of XSS and SQL injection attacks. Its performances are acceptable but requires a learning process to define white-lists. SQLi and XSS rules are static and use a simple cumulative scoring system based on the appearance of some special strings.

III. SQLi ATTACKS AND EVASION TECHNIQUES

Halfond, Viegas, and Orso researches [5] proposed a taxonomy of SQL injection attacks. Depending on the goal, attackers can append a syntactically correct SQL code to the original query, or forge their own malicious SQL commands and introduce them to the DBMS via a vulnerable web application inputs. They will use one of these classes: *Tautology, Incorrect Queries, UNION Queries, Piggy-backed Queries, Stored Procedure, Inference (Blind Injection, Timing Attacks).* In the following, we will present two techniques attackers can use in order to bypass the WAF rules.

A. SQL injections by code obfuscations

Despite the deployment of SQLi attacks detection systems, attackers can manage to overcome these systems by trying to disguise the appearance of their requests. This is due to both the flexibility of the SQL language and the complexity of writing security rules wich express all scenarios of a masked attack. The combined use of ASCII encoding, Hexadecimal, and trans-coding functions, which are available by the CGIs, makes the pattern matching inspection process more complex. Indeed, the pattern matching algorithms are able to detect variations in the forms but they can't detect trans-coding of string characters. Thus, attacks detection systems start with the transformation (sanitization) of the query before submitting it to the pattern matching engine. Despite these efforts, it is always possible to bypass filters posed by these systems, because both the flexibility of DBMS languages and the semantic content of the query are not known by the detection systems. More details are shown in the following example :
- Uppercase/lowercase blend: Some filters often ignore the situation of mixed case, such as the input AND or and which can be replaced by AnD
- Tautology: (where name = '='), (where false = " "), (OR true like true)
- SQL keywords: SELECT becomes sel/**/ect , AND becomes && or AandND
- The use of trans-coding functions : (char(44) = ') , (exec(char(Ox73687574646j776e)) is a system shutdown.

B. Headers-level SQL injection

The injection of SQL codes is not only restricted to GET and POST methods through URLs or request's body. Unfortunately, other more injection techniques exploit the open paradigm of the HTTP protocol in order to avoid web applications' security mechanisms. Attackers exploit HTTP protocol headers because most defense systems analyze only few or probably do not analyze HTTP headers due to the huge resources requirement.

1) User-agent Header: This header is used by web applications to store information about the client application (web browser, audit application, etc). Some e-commerce applications store this information in databases for clients profiling. An attacker can exploit this header to insert SQL code that overcome security checks. He can insert a tautology in order to make the check condition always valid. Example : user-agent : Firefox' OR 1=1.

2) Referer Header: Referer is another HTTP header which can be vulnerable to SQL injection once the application is storing it in database without sanitizing it. An attacker may inject arbitrary SQL commands to the database by exploiting the Referer header. With the same way as in the User-Agent, the attacker can insert a tautology to make the control over the completely useless Referer or insert SQL malicious code after separator:
Referer: 1','0'); SQL

3) X-Forwarded-For Header : This header is used to get the original IP address of a client who is connected through a proxy or other load-balancing device. An attacker can also make the control obsolete by inserting a tautology. Example : X-Forwarded-For :ip-address or 1=1

4) Cookie Header : Web applications use the cookie header in order to ease the authentication process of their clients. It often happens that the application does not validate the cookie passed by the client and inserts it directly in an SQL statement that selects the user session. An attacker can then inject SQL code in this header and completely avert the application authentication mechanism. He can use tools such as cookie manager to forge a valid cookie. Cookie variables sometimes are not properly sanitized before being used in SQL query. The cookie contains base64 encoded form identifier, a field that is unknown and a password. If we use as a cookie 12345 UNION SELECT mypass :: mypass base64 encoded, the SQL query becomes:
SELECT user-password FROM nk-users WHERE user-id=12345 UNION SELECT mypass

IV. Proposed Solution

A. HTTP Protocol Dissection

In most security solutions, traffic dissection process is the first operation before applying any security control. In HAKA project [10], HTTP stream is divided in two tables, HTTP-request and HTTP-response, each table is associated with one hook and each hook contains all security rules declared for either requests or responses. In this section we introduce an other manner to dissect HTTP streams.

a) Typical HTTP request: HTTP protocol is expressed in a human-readable ASCII text. Headers use text to describe a request from a client (browser) or a response from the server. An HTTP request begins usually with a GET or POST method, followed by the URL and the protocol version. The following headers provide various information about the client, connection, content, etc. These headers are separated by $\backslash r \backslash n$ to distinguish each header.

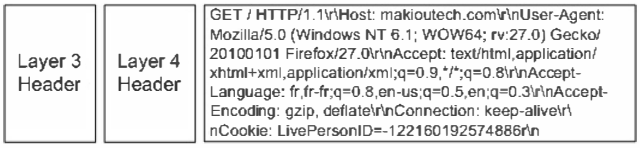

Fig. 1: Raw HTTP Request

b) HTTP Request Dissection: Our dissection module is able to recognize request's components (headers and the body) which are separated by $\backslash r \backslash n$ characters. However, before making the dissection, it has to get information about security rules. Indeed, users are obliged to declare security rules for the body and for each header. With the knowledge of headers involved in the inspection process, the dissector will only extract and parse these headers.

B. The Hybrid Injection Prevention System (HIPS)

• 1 Request Dissector URL: The information that the URL must be inspected is known by the dissector, since the user has declared a security rule on the hook : *HTTP URL*. The dissector extracts URL string and passes it to the classifier.
• 2 If legitimate: The result of the classification is negative, which means that the URL string does not include any potential SQL injection code.
• 3 If SQLi: The classifier decides to forward the URL string to the security rules detection engine, because the URL contains probably an SQL injection code. In the result section, we will show the minimum threshold for which a legitimate content is considered suspicious.
• 4 If SQLi: The detection engine loads all the security rules hooked to the *HTTP URL* hook
• 5 If matched: The detection engine uses a pattern matching algorithm to inspect the URL content, if one security rule matches with a specific pattern, the detection engine rejects the HTTP request.
• 6 No Rule matched: After applying all security rules, no rule matches with the analyzed content.

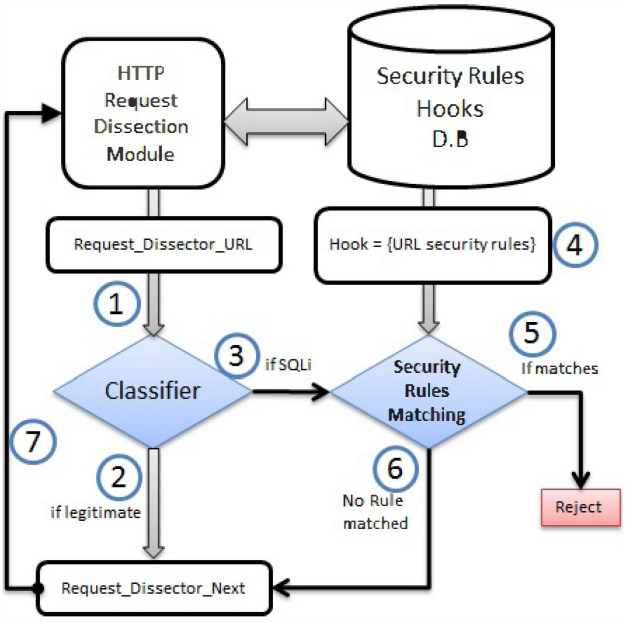

Fig. 2: Detection Engine Architecture

• 7 HTTP Next Dissector: The header analyzed above doesn't contain a malicious code, next headers will be analyzed in the same way until the end of dissectors.

C. Data Collection and Representation

1) SQL injections data collection framework: Malicious traffic is collected from an attack platform that includes:
- Specific SQLi attack tools using evasion techniques
- A Web server
- A vulnerable web application with known SQL injection attacks.

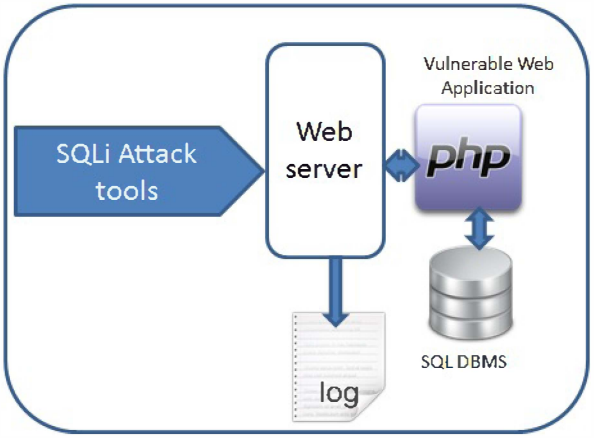

Fig. 3: Attack Traces Collection Framework

2) features vector selection: Feature selection plays an important role in the identification of the potential malicious data used to escape SQLi filters. Based on our experience with

SQL injections attacks and evasion techniques, we identify a limited vector of features. There are several motivating factors behind limiting the feature set of both the SQL injection and evading strings for our classifier. A smaller feature set may result in a significant decrease of both learning and classification time. The following table shows the result of our features vector selection.

Variables	Tokens (key words)
x1	SELECT
x2	UNION
x3	UPDATE
..	Other SQL keywords
xn	* (Asterisk)
..	Other Special Symbols
xm	UNHEX
..	Other Evasion Keywords
x45	% (percent)

TABLE I: Features Table

3) Request representation: Each request's header is characterized by a vector $\overrightarrow{x}$ defined by $\overrightarrow{x} = (x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are the values taken by the random variables $X_1, \ldots, X_n$ that are assumed conditionally independent relative to the category c (SQLi, Legitimate). Each random variable gives an information about a pattern type in a dissected header. In this model, all random variables are binary: $X_i = 1$ if pattern of type i noted pa_i is present, otherwise $X_i = 0$. Consequently, each random variable $X_i = 1$ follows a Bernoulli distribution with parameter $p_i = p(pa_i)$.

D. The Machine Learning Model

A naive Bayesian model, which we have adopted in our classifier, is a simple classification scheme that estimates the class-conditional probability by assuming that features are conditionally independent. In fact, we have a binary classification problem of identifying the normal http stream as legitimate requests class and malicious stream as SQLi requests class. According to the Bayes theorem [11] and the total probabilities theorem, for a vector $\overrightarrow{x} = (x_1, \ldots, x_n)$, the probability to belong to the class c is defined as follows:

$$p(C{=}c|\overrightarrow{X}{=}\overrightarrow{x}){=}\frac{p(C{=}c){\cdot}p(\overrightarrow{X}{=}\overrightarrow{x}/C{=}c)}{p(\overrightarrow{X}{=}\overrightarrow{x})} \quad (1)$$

Using the theorem of the total probabilities, we deduce:

$$p(C{=}c|\overrightarrow{X}{=}\overrightarrow{x}){=}\frac{p(C{=}c){\cdot}p(\overrightarrow{X}{=}\overrightarrow{x}/C{=}c)}{\sum_{c\epsilon\{SQLi,Leg\}} p(C{=}c){\cdot}p(\overrightarrow{X}{=}\overrightarrow{x}/C{=}c)} \quad (2)$$

1) Cost Evaluation: Works of [12] on cost-sensitive evaluation measures consist of evaluating the false positives effect on the total cost, in term of time wasting by users to delete spams. But in our case, the cost is the impact of false negatives on the trust of users granted to our classifier.

2) The False Negatives Effect: A false negative is mistakenly classifying an SQLi attack as a legitimate content, and a false positive is a legitimate content mistakenly classified as an SQLi attack. In our model, the cost of a false negative is much higher than the cost of a false positive. Indeed, wasting time in analyzing legitimate requests is more acceptable than passing a malicious code to the Web application. The two error types are defined as such follows:

- Classifying an SQLi attack as legitimate content: $(SQLi \rightarrow Leg)$
- Classifying a legitimate content as an SQLi attacks: $(Leg \rightarrow SQLi)$

In our classifier model, the first error is more serious than the second error. To illustrate this idea, we introduce the parameter λ, as its objective is to give more importance to the first error by assuming that $Leg \rightarrow SQLi$ is λ times more costly than $SQLi \rightarrow Leg$.

3) Classification criteria: According to the above two error types, the selection criteria is as follows:
The content of a header $\vec{x}$ is classified as legitimate if and only if:

$$p(C = Leg|\vec{X} = \vec{x}) > \lambda.p(C = SQLi|\vec{X} = \vec{x}) \quad (3)$$

given that $p(C = Leg|\vec{X} = \vec{x}){+}p(C = SQLi|\vec{X} = \vec{x}) = 1$, the selection criteria becomes as follows:

$$p(C = Leg|\vec{X} = \vec{x}) > \alpha \quad (4)$$

Where :

$$\alpha = \frac{\lambda}{1+\lambda}, \lambda = \frac{\alpha}{1-\alpha} \quad (5)$$

4) Method and parameters evaluation: In this section, we define the parameters that allow us to evaluate our filter. To this end, two evaluation parameters are used: accuracy (Acc) and error ($Err = 1 - Acc$) [12]. They are defined as follows:

$$\begin{aligned} Acc &= \frac{n_{sqli\rightarrow sqli} + n_{leg\rightarrow leg}}{N_{sqli} + N_{leg}} \\ Err &= \frac{n_{sqli\rightarrow leg} + n_{sqli\rightarrow sqli}}{N_{sqli} + N_{leg}} \end{aligned} \quad (6)$$

where:

- $N_{sqli} = n_{sqli\rightarrow leg} + n_{sqli\rightarrow sqli}$
- $N_{leg} = n_{leg\rightarrow leg} + n_{leg\rightarrow sqli}$
- $n_{y\rightarrow z}$ denotes the number of patterns of class y that are mistakenly classified in class z.

The parameters defined above do not take into consideration the notion of weight for the two error types introduced in the previous paragraph. This leads us to introduce the weighted accuracy ($Wacc$) and weighted error ($Werr = 1{-}Wacc$). We assumed that $SQLi \rightarrow Leg$ is λ times more devastating for our system than $Leg \rightarrow SQLi$. To make accuracy and error

rate sensitive to this cost, we should treat each SQL injection as if it was λ inputs; when an SQL injection is misclassified, this counts as λ errors; and when it is classified correctly, this counts as λ successes

$$Wacc = \frac{\lambda n_{sqli \to sqli} + n_{leg \to leg}}{\lambda N_{sqli} + N_{leg}} \qquad (7)$$
$$Werr = \frac{\lambda n_{sqli \to leg} + n_{leg \to sqli}}{\lambda N_{sqli} + N_{leg}}$$

To have a precise idea of the filter's performance, we compare it to a non-filtered system in which all requests are considered as legitimate.
We introduce the definition of the base-line weighted error and the base-line weighted accuracy (respectively noted $Wacc^b$ and $Werr^b$) which are defined as follows:

$$Wacc^b = \frac{\lambda N_{leg}}{\lambda N_{sqli} + N_{leg}} \qquad (8)$$
$$Werr^b = \frac{N_{sqli}}{\lambda N_{sqli} + N_{leg}}$$

The TCR (Total Cost Ratio) value measures the performance of a machine learning classifier to the same environment without a classifier. In the case where the TCR value is negligible, the best approach is to not use a classifier and to send all requests to the rules based detection engine. An effective filter which could be used in real environments should have a TCR value higher than 1.

The TCR formula is defined as follows:

$$TCR = \frac{Werr^b}{Werr} = \frac{N_{sqli}}{\lambda n_{leg \to sqli} + n_{sqli \to leg}} \qquad (9)$$

V. RESULTS EVALUATION

We have collected a training data set from the framework previously presented in this paper. The training set is a mixture of SQli attacks and legitimate requests shown in the below table with different proportions (80-20)%,(66-34)%,(50-50)%, then we varied, for each scenario, the λ parameter and finally calculated values of $Wacc$, $Wacc^b$ and TCR.

By increasing the α (threshold) value from 99% to 50%, we have an increase in number of false positives, which means that the rules engine will analyze legitimate requests, but at the same time, the evaluation has shown an increase in TCR value. However, in practice, false positives (legitimate requests classified as SQLi attacks) will be forwarded to the rules engine that will apply all security rules. This will decrease the overall performances of our system. To find a compromise between lower number of false positives and false negatives, the TCR values should be medium values. Acceptable values of TCR are related to α threshold values higher than 50% and lower than 90%.

Data (%) SQLi-Leg	λ Num	α (%)	False Pos.(%)	False Neg.(%)	TCR Value
80-20	1	50	4.6	0.6	15.38
66-34	1	50	2.0	0.3	28.57
50-50	1	50	3.0	0.5	14.29
80-20	2	66.67	6.0	0.5	8.16
66-34	2	66.67	2.0	0.3	15.38
50-50	2	66.67	3.0	0.5	7.69
80-20	5	83.33	5.6	0.4	3.39
66-34	5	83.33	2.9	0.3	6.45
50-50	5	83.33	4.0	0.6	3.23
80-20	9	90	5.8	0.6	1.90
66-34	9	90	4.0	0.3	3.54
50-50	9	90	5.0	0.5	1.82
80-20	99	99	6.6	0.4	0.18
66-34	99	99	4.0	0.3	0.34
50-50	99	99	5.0	0.4	0.17

TABLE II: Costs Table

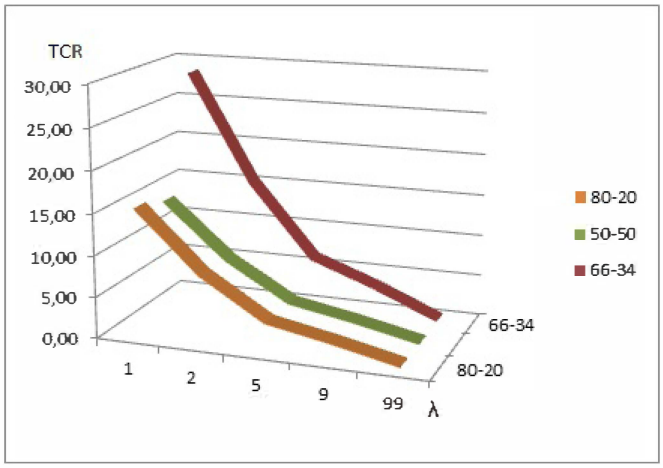

Fig. 4: Total Cost Ratio

VI. COMPARATIVE ANALYSIS

Security Rules based WAFs do not have a module that predicts SQL injection attacks. In this case, our approach will improve the inspection performance because the classifier module will not forward legitimate traffic to the detection engine. Only suspected requests are deeply inspected by applying pattern matching algorithms. In [13], Authors worked on *Classification of Malicious Web Code by Machine Learning*. They implemented and evaluated two classifiers both SQLIAs and XSS which can use TF-IDF method for weight calculation, and three machine learning approach among SVM, Naive-Bayes, k-Nearest Neighbor Algorithm. They obtain good precision values for their classifier (99.16 %) by using SVM with Gaussian Kernel. Our classifiers obtains 97.6% by using Bayesian algorithm. However, they do not provide a solution to handle false negatives and false positives. On the other hand, the use of SVM algorithms with Gaussian Kernel may require significant CPU resources and decreases the classifier performances in multi-gigabits rates networks.

VII. Conclusion

In this paper, we focused on the problem of detecting complex SQL injections. We proposed a novel approach to dissect HTTP requests in order to cover most evasion techniques and improve security rules management process. We also provided an Injection Prevention System architecture which includes a machine learning classifier. Based on the TCR results, we have shown the effectiveness of the classifier by tuning its values in order to reduce false negatives. We were also able to show that false positives did not impact the overall performances of our system.

A key element of future work is to apply the same approach in order to develop an anti XSS and SQL attacks solution.

References

[1] The Open Web Application Security Project (OWASP) considers, in its 2013 top ten list. Available: $https://www.owasp.org/index.php/Top_10_2013-Top_10$

[2] Sid Ansari et al. SQL Injection in Oracle: An exploration of vulnerabilities International Journal on Computer Science and Engineering (IJCSE), pp. 522-531, April 2012

[3] A.Tajpour, M. Massrum and M.Z. Heydari, Comparison of SQL Injection Detection and Prevention Techniques, 2nd International Conference on Education Technology and Computer (ICETC), 2010

[4] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. 10th ACM Conference on Computer and Communication Security (CCS 03), pages 251261. ACM Press, October 2003

[5] W.G. Halfond and A. Orso, AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks, Proc. 20th IEEE and ACM Intl Conf. Automated Software Eng., pp. 174-183, Nov. 2005

[6] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, pages 292-302. June 2004

[7] Ivan Ristic : ModSecurity Handbook: The Complete Guide to the Popular Open Source Web Application Firewall, 2010 Feisty Duck Ltd Edition ISBN: 1907117024

[8] The IronBee Project May 2014. Available:$http//www.ironbee.com/$

[9] Naxsi project (Nginx Anti Xss Sql Injection) May 2014. Available: $https://www.owasp.org/index.php/OWASP_NAXSI_Project$

[10] Kevin Denis, Pierre Sylvain Desse et Mehdi Talbi (Arkoon Network Security), un langage orient rseaux et scurit, Symposium sur la scurit des technologies de l'information et des communications,Confrence francophone sur le thme de la scurit de l'information, 2014.

[11] C.P.Robert,Le choix Baysien. Principes et pratiques, Ed. Springer,2006

[12] Androutsopoulos I., J. Koutsias, K.V. Chandrinos, G. Paliouras, and C.D. Spyropoulos.2000a. An Evaluation of Naive Bayesian Anti-Spam Filtering. Proceedings of the Workshop on Machine Learning in the New Information Age, 11th European Conference on Machine Learning, Barcelona, Spain, pages 917.

[13] Komiya, R. Incheon Paik Hisada, M.Classification of malicious web code by machine learning.Awareness Science and Technology (iCAST),011 3rd International Conference. Sept 2011.