Achieving Web Security by Increasing the Web Application Safety

Maryam Abedi
Dept. of Information Technology Eng., Shiraz University, Shiraz, Iran
maryam_abedy@yahoo.com

Navid Nikmehr
Dept. of Information Technology Eng., Shiraz University, Shiraz, Iran
nikmehr@ieee.org

Mohsen Doroodchi
Dept. of Math/Computer Science,Cardinal Stritch University, Milwaukee, WI
mdoroodchi@stritch.edu

***Abstract*—As web applications have become an integral part of today's business operations, the concerns about the security of exchanged information on the web have been increasing. Issues such as data breach and leakage of sensitive information is number one concern of businesses for which the web applications are blamed for the most part. Therefore, in addition to the common measures used to secure the communications and transactions on the web, more attention needs to be paid to the preventive measures of integrating security into the development phase. However, for evaluation of effectiveness of such measures, a quantitative method is very essential to calculate the safety of an application against different vulnerabilities. This work presents a new model for measurement of overall safety of web applications. The keyword "safety" is coined to distinguish this measure from the traditional methods.**

Keywords: quantitative measurement,web application security, safety, vulnerability

I. INTRODUCTION

Enterprises' critical resources are highly in risk of cyber-attacks due to the vast delivery of enterprise applications with vulnerabilities over the web. Reports of the catastrophic hacking stories reveal that the sensitive data are compromised through web application vulnerabilities. In order to deliver safe applications over the web, such vulnerabilities are required to be studied and understood in depth.

Different forms of injections are reported to be the major concern with web applications. According to OWASP [25] and SANS 2011 Top 25 [41], SQL injection is ranked first among web application vulnerabilities. After that, code and shell injection are introduced as the second security issue in today's' web applications. Based on a report by Viega and McGraw, code injection is known as the most challenging security breach as a result of poor input sanitization [39].

Despite considerable research on understanding and managing the security issues, including qualitative aspect of security measurement such as BS7799 [4, 5], ISO17799, NIST SP800-33 [2, 29, 30], there are only few quantitative metrics [3] available for measuring security related issues. These methods are often either not comprehensive enough [19, 22] or are limited only to their specific measurement model which reduces the usability of the model and some are too complicated to be used by developers [26].

This is an undeniable consensus that the capability of measuring, comparing, and contrasting different entities provides the opportunity of a thorough understanding of the underlying concept [22] as Lord Kelvin in 1883 stated: "*When you can measure what you are speaking about and express it in numbers you know something about it, but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind*". And security management of web applications through measurement is not an exception.

In this paper, we overlook the concept of enterprise application security in terms of a quantifiable concept we coined as web application safety. The proposed measurements model is using the known vulnerabilities and at the same time is scalable to use the new vulnerabilities. Our aim is to find a practical and universally acceptable quantitative model that can be integrated into the software development life cycle. The proposed model allows the developers to measure the safety of their under application during different development phases.

The foundation of our model is based on measurement of two aspects of standard and best practice prevention methods integrated into the projects. These two aspects are called efficiency and sufficiency of the methods which are explained in details later. In addition, we quantify the effectiveness for each method. At this time, the coefficient of effectiveness is determined subjectively based on experts' perspective.

Since the discovered vulnerabilities are rapidly increasing, the capability of appending additional vulnerabilities to our model along with their corresponding mitigation methods, leads to enhance the flexibility and extensibility of the proposed measurement model. In addition, this extensibility feature provides flexibility to redefine, modify, and improve the proposed quantification metric definitions throughout the development process.

The rest of this paper is organized as follows. Next section covers the proposed model for web application safety measurement followed by proposed metrics used in the overall calculations followed by the experimental results. The last section of this paper is the conclusion.

It is notable to mention that quantified level of safety against specific vulnerabilities for any application is specific to that application and it cannot have a specific scale and range. Therefore, this method is best fit for comparison of the safety level for different versions of the same application.

II. PROPOSED SAFTEY MODEL

As mentioned, this model starts from a list of known vulnerabilities and the corresponding measurement metrics to calculate the overall safety of a web application. To achieve this task, we propose a hierarchical model as illustrated in figure 1. The metrics are categorized based on different vulnerability types as depicted in this figure. Furthermore, in order to evaluate the model we select different metrics to measure the safety of a web application against SQL injection and Shell injection as explained in the following sections. We chose six metrics for each category. Each metric evaluates the sufficiency (and/or) efficiency of

possible preventative method that could have been implemented to raise the overall safety of the application.

The overall safety of an application can be visualized as the root of a tree as shown in Fig. 1 in which the branches are providing the particular safety measurement for a given preventative method.

This approach has a number of benefits. First, the security tester can plan the test using different combination of available vulnerabilities. Second, these metrics reveal the interdependencies of different vulnerabilities to the developers, and consequently the application developer could provide additional isolation between them.

Moreover, this model is scalable and flexible to add new metrics for known or new vulnerabilities. Next section explains each metric in details. For this purpose, we consider the following two parameters for each metric, 1) a name, and 2) a description. The description provides information about the vulnerability and corresponding mitigation methods which can increase safety. It also provides a proposed formula that measures the sufficiency and/or efficiency of mitigation method and returns a numerical value. Each mentioned formula needs some inputs – aka vector of inputs- to return the safety value. We also define two properties for each member of this vector; input name, description description and a numerical value.

This numerical value can be entered by the user of the model based on the application or comes from another formula's numerical result of other metrics. Clearly, larger values of the results of each of the formula would contribute directly to a safer application which consequently results in an overall increase of security.

In addition, this model has three more parameters to achieve more accurate value for overall safety. These parameters are listed as follows. The *first* parameter is "*e*", the effectiveness coefficient, as shown in Fig. 1. It is clear that all the mitigation mechanisms do not have the same contribution toward the application safety improvement. With respect to this fact, this parameter reflects the metric's relative importance and effectiveness in mitigating the overall vulnerability of the application. Obviously, the ones that are more effective have greater weight. We recruit fuzzy logic to determine effectiveness coefficients' value. The process of determining this value is fully explained in section V.

The *second* parameter is "score of vulnerability" as shown in figure 1. It is clear that all types of vulnerabilities do not endanger the safety of a web- application equally; hence this parameter reflects the weight of the vulnerability. In this work, its value equals to the score of the vulnerability in CWE/SANS ranking system, as a reputed reference in this area. The vulnerabilities are prioritized and scored according to their prevalence, importance, and likelihood of exploiting [8].

The *third* parameter is "*Phase of lifecycle*" as depicted in Fig. 1. Our model is capable of evaluating a web application safety in any of the main three phases of analysis, design, or implementation phase of SDLC. This parameter should be applied as a consistent value throughout the safety evaluation process. Given the fact that implementing of any prevention method in the earlier phases has more effectiveness than postponing them to next phases [13], we assign a greater coefficient to earlier phases. Note that the formulas of metrics for one phase are different with another phase, but the general principle of the model is the same. On the other hand, obviously, the value for all parameters of the model should be obtained from the same phase.

Based on the above explanations, the overall safety against a specified vulnerability is demonstrated as *OSAV* function as follows.

$$\boldsymbol{OSAV} = c * \sum_{i=0}^{n} (f_i(p)) \qquad (2)$$

$$f(p) = a * \sum_{j=0}^{n} (e_j * p_j) \qquad (2)$$

, where c is score of vulnerability based on "CWE /SANS" [8], and a represents phase of lifecycle of given project, and e is effectiveness $\in$ [0,1], and p represents result value of quantification formula for evaluation of specific preventative method.

As mentioned before, in this work we examine our model and its metrics for the first two highest ranked vulnerabilities as mentioned in the 'Top 25 Common Weaknesses Enumeration (CWE) database' [1]. The database that is sponsored by Mitre, is used frequently as a reference by application developers and security engineers to identifying possible weaknesses to attack in software applications. However, it does not mean that this model is restricted to assess safety against this database's vulnerabilities. The proposed associated quantitaitive formulas for mentioned vulnerabilities are defined in next section.

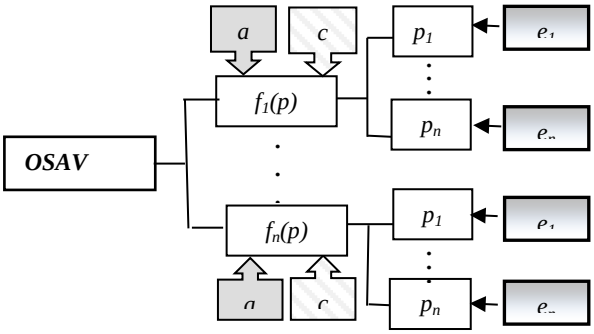

Fig.1.The proposed model

III. Proposed Metrics

In this section, we explain the details of proposed quantitative metric for the top-two web application vulnerabilities from "Top 25 Common Weaknesses Enumeration (CWE) database" [1] to evaluate the sufficiency and/or efficiency of common (and standard) preventive mechanisms in a given application.

A. SQL Injection Vulnarability

According to [35], "*SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQL Server will execute all syntactically valid queries that it receives.*" SQL Injection vulnerabilities represent about 20% of reported vulnerabilities recorded in commonly availablevulnerability databases[1] as CWE/SANS assign the score of 93.8 to this vulnerability [8]. Therefore, safety against SQL injection is very critical to web applications. The following list introduces metrics for quantifying safety of an application against SQL injection.

1) Type of Inputs used on the Forms

- *Name*: SQLInj-001- Form Field Type

[1] www2.cenzic.com/downloads/Cenzic_AppSecTrends_Q1-Q2-2010.pdf

- *Description*: When textboxes are used to get the user inputs, they can also be used by dynamic SQL queries to boost the danger of SQL injection attacks [37]. Thus, more textboxes for inputs/outputs, more chance of possible dynamic SQL queries which leads to less safety against SQL injection. We propose the efficiency function $f(n_1, n_2)$ as the ratio of total number of inputs to the number of textboxes as shown below.

$$f(n_1, n_2) = = \begin{cases} n_1/n_2 & n_2 > 0 \\ n_1 & n_2 = 0 \end{cases} \quad (3)$$

, where n_1 is total number of form fields such as textboxes, radio buttons, checkboxes, dropdown menus, etc., and n_2 represents total number of textboxes in project's forms that collect and send user's data to dynamic SQL statements.

2) Error Presentation Mode

- *Name* : SQLInj-002-Error presentation Mode
- *Description*: due to default database management system behavior of throwing error messages, attackers can potentially expose the structure of databases and It is obvious these error messages help attackers to get a hold of the information which they are looking for (such as the database name, table name, usernames, password hashes etc.). As a mitigation strategy, a particular generic or specific error message should be used in error susceptible cases [36, 18]. To assess the potential database exposure through error messages, wedefine $f(n_1, n_2)$ to measure the sufficiency of error exception handling of application as follows.

$$f(n_1, n_2) = \begin{cases} n_1/n_2 & n_2 > 0 \\ n_1 & n_2 = 0 \end{cases} \quad (4)$$

, where n_1 is total number of exception handling mechanisms implemented in the project and $n_2$is total number of scenarios that are prone to throw default error message. As we mentioned in previous section , the more value this equation has, the error exception handling -as a mitigation strategy - has been performed the better.

3) Input Validation

- *Name: SQLInj-003* - Input validation
- *Description*: The common weakness that can make an application susceptible to SQL injection is weak input validation. These inputs normally include form data, URL parameters, hidden fields, cookie data, HTTP Headers, and essentially anything in the HTTP request [32].

Constraining input for type, length, format, and range [32], filtering meta characters such as beginning of a comment character, or characters that denote the end of one query or the beginning of a SQL statement [28] are useful strategies to validate data and prevent SQL injection.

As previously mentioned, default error messages may expose the structure of database. An attacker can penetrate into the database by trying particular SQL commands. Accordingly, SQL statements that are used to retrieve or manipulate data are better to be filtered [38].

One of the most common validating strategies to increase security is recruiting range validation technic and also data validation based on matching with a proper regular expression. In other words, we ought to use the approach "Accept Known Good" instead of "Reject Known Bad" [33].

The following function evaluates the sufficiency of validation strategies in an application.

$$f(n_1, n_2) = \begin{cases} (n_1 + n_2 + n_3 + n_4 + n_5 + n_6 + n_7)/m & m > 0 \\ 0 & m = 0 \end{cases} \quad (5)$$

,where m is total number of any data input that are thrown to dynamic generated SQL statements including form data, URL parameters, hidden fields, cookie data, HTTP headers, and any piece of data in HTTP request [32], n_1 is total number of any *data type* validation, n_2 is total number of any *data format* validation, n_3 is total number of any *data range* validation, $n_4$is total number of any SQL meta-characters that has been filtered, n_5 is total number of any SQL commands that has been filtered, $n_6$is total numbers of regular expression validators, and $n_7$represents total number of range validators that have been recruited in application.

4) SQL Statement Generation Mode

- *Name*: SQLInj-004 -SQL statement generation mode
- *Description*: SQL Injection flaws are introduced by utilizing dynamic queries. In this scenario SQL statements are generated based on user's input and each user could be potentially an attacker. Therefore, implementing more dynamic database queries in an application makes it more vulnerable to SQL Injection [34]. The function $f(n_1, n_2)$ measures the efficiency of SQL statement generation approach, by ratio of total number of SQL statements as numerator and total number of dynamical SQL statements as denominator.

$$f(n_1, n_2) = \begin{cases} n_1/n_2 & n_2 > 0 \\ n_1 & n_2 = 0 \end{cases} \quad (6)$$

,where n_1 is total number of SQL statements including static and dynamic ones, and n_2 is total number of SQL statements that are generated dynamically.

5) Efficieny of stored procedure

- *Name*: SQLInj-005-Efficient utilization of srored procedure
- *Description*: A stored procedure is a group of SQL statements that has been created and stored in the database [15].To boost safety agnist SQL injection flaws, use of stored procedure is highly recommended as long as they do not include any unsafe dynamic SQL generation [11]. Not only the number of implemented stored procedure increases the application safety, but also their performance contribute to more safety.Thus, here we use the aforementioned input validation sufficiency function as well as the SQL statement generation efficiency function to measure the efficiency of implemented stored procedures. Furthermore, since more stored procedure implementation increases application safety against SQL injection, considering total number of stored procedure is also required. Therefore, $f(a,b)$ is used to assess the competence of stored procedue engagment in software.

$$f(a,b) = \begin{cases} \sum_{i=0}^{n} (a_i + b_i) & n > 0 \\ 0 & n = 0 \end{cases} \quad (7)$$

, where a_i is value of "Input Validation" metric function (SQLInj-003) of the (stored procedure)$_i$, $b_i$is value of the "SQL Statement Generation"

metric function (SQLInj-004) of the (stored procedure)$_i$, and n represent total number of stored procedures.

6) Access Restriction

- *Name*: SQLInj-006 -Access restriction
- *Description*: Safety against SQL injection attacks is related to how many users access to how much of data and more exposure of data in term of number users that access it causes application to be more vulnerable.

Hence, to measure the efficiency of access restriction policy in application, it is required to define that each level of access permission is granted to how many users as the *first* variable. Each level's access permission should be specified also to reflect the accessible data through that particular level, as the *second* variable. Our assumption is that the greater value for level of access permission corresponds to higher access permission.

Therefore proposed $f(n)$ calculates its value by multiplying abovementioned variables for each level. Then summarize the products values of all implemented levels.

Obviously, the greater value of each multiplication (and following that the summarized value) implies the less imposition of access restriction policy which has an adverse effect on safety. Since safety against SQL injection $\propto$*1/data accessibility*, we propose $f(n)$ as follows:

$$f(n)=1/\sum_{i=0}^{m}(n_i*i) \qquad (8)$$

, where n_i is total number of granted accesses to level i, and m is total number of access levels which has been defined in application.

B. OS Command Injection

Briefly, applications are considered vulnerable to the OS command injection also known as shell injection attack if they utilize user input in a system level command. Shell injection attacks lead to execute risky commands on operating system through an application when the attacker does not have direct access to OS. Alternatively, it may make a number of OS restricted commands accessible for attacker when application is privileged [6]. This vulnerability mostly happens when there is an under control procedure in application which needs externally-supplied input arguments to be executed and/or when there is the possibility of getting the externally-supplied procedure or commands calls.Then entire given command has been sent directly to OS for ecxecution [9, 24].According to CWE/SANS findings, the score for this vulnerability is 83.3 [8].

In this section, a number of common preventative methods to raise safety of a project against shell injection compromise are introduce and then corresponding metrics are proposed to quantify the sufficiency and/or efficiency of thoes methods.

1) Function type generation

- *Name*: ShellInj-001- Function type generation
- *Description*: To boost control on input data, it is recommended to recruit library call policy, instead of using external proccess [9]. We quantify sufficiency of using library calls by calculating the ratio of number of library calls in the application as numerator, over total number of library call plus external processes as denomirator through $f(n_1, n_2)$. Obviously, the greater numerator value means more satefty provision.

$$f(n_1, n_2)=n_1/n_2+ n_1 \qquad (9)$$

, wheren_1 is total numbers of third-party libraries that are called to generate functions, and n_2 is total number of external processes recruited to generate functions.

2) Jail Or Sandbox Utilization

- *Name*: ShellInj-002- Jail and Sandbox Utilization
- *Description*: Itis recommended to enforce strict boundaries between process and operating system.This may restrict which data can be accessed or which commands can be executed by application.

However, this solution may only limits the impact to operating systembut rest of the application may still be subject to compromise [9]. A possible solution for such enforcement is to utilize sandbox enviroment.

A sandbox is a security mechanism for separating running programs and quarantining untrusted running programs. It can be used to execute untested code or untrusted programs from unverified third-parties, suppliers, untrusted users and untrusted websites [16]. Jail sets is a common strategy of sandbox mechanism. Jail is a set of resources limits imposed on programs by operating system kernel (e.g. I/O bandwidth caps and disk quotas) [16,10].The effectiveness of this method depends on the deterrence capabilities of the particular sandbox or jail.It may only reduce the scope of an attack, such as restricting the violator to excecute certain system commands or limiting the data that can be accessed. We demonstrate $f(a_1,a_2)$ by imposing a logical *OR* function on thoese above mentioned strategies recruitment.

$$f(a_1 a_2)= (a_1)OR(a_2) \qquad (10)$$

, wherea_1 is "code runs in jail sets" as a boolean variable ($a_1 \in \{0,1\}$), and a_2 is "code runs in other forms of sandbox environment" as a boolean variable and $a_2 \in \{0,1\}$.

3) Input Validation

- *UniqueID*: ShellInj-003≡SQLInjection.SQLInj-003 (Input Validation)
- *Description*: It is highly recommended to validate input since it has a deterrent effect for OS command injection, [21,31]. we quantify its sufficiency and efficiency in the same that has been disccuesed in SQLInj-003 metric in previous section.

4) Error Presentation Mode

- *Name*:ShellInj-004≡SQLInjection.SQLInj-002:Error presentation Mode
- *Description*: Stephanie Reetz considered system defualt error messages as informative and precious data for adversaries thatraisesthe risk of shell injection compromise, [36]. Therefore the more managed error messages is implemented in application, the risk of penetration will be declined. We quantify sufficiency of implemented error exception handeling mechanisms same way that has been discussed in SQLInj-004 metric in previous section.

5) Accounts Isolation

- *Name*: ShellInj-005-Accounts isolation
- *Description*: In order to mitigate shell injection breaches, it is recommended to create role-based access control scheme with restricted privileges in order to be used only for a group of specified tasks and users. By following this strategy, a successful attack will not accompolished because the rest of application or its environment is not accessible to attacker [40, 23].

We quantify the efficiency of implemetation of this policy by means of a linear function. $f(n_1,n_2,n_3)$ is the ratio of specified roles in

application over the total number of critical tasks plus critical resources. The more specific roles is defined (i.e greater numerator value) to access and excute respectively critical resources and tasks, leads toincrease theefficiency of account isolation policy.

$$f(n_1, n_2, n_3) = n_1/(n_2 + n_3) \quad (11)$$

, where$n_1$is total number of access roles that are specified to access critical resources and execute critical procedures, n_2 is total number of critical resources, and n_3 representstotal number of critical procedures

6) Reduction of Attack Surface

- *Name* : ShellInj-06-Attack Surface value
- *Description*:It is obvious that the more resources are available to users, the more exposed to attacks the application is [27, 12]. In [20], the authors argues that the attack surface of an application environment is the sum of the different waysthat an attack action can perform through them in order to enter or extract data from an environment. They measure attack surfaceby means of quantifying the application's interaction with its environment through three types of recourses;entry/exit points, channels, and untrusted data items. According to them, in order to measure the attack surface,we should evaluatethe probability that an adversary will use each specific resource in an attack. This evaluation is also accomplished in another work using the ratio of the potential damage-termed *Damage/effort Ratio*(DER) [14], where damage corresponds to possible technical advantages of that resource,and effort isthe amount of effort needed to access that. The value for effort in this ratio can be derived from level of access rights that isneeded to access that specific resource. The final measurement formula is expressed in a triplet of three DERs of three mentined resouces types includes entry/exit points, channels and un-trusted data items[14]. The greater value of this triplet implies more damage for a consistent effortvalue, which means greater attack surface value as an application's weakness. To utilize this triplet to measure attack surface, we have to slightly modify it for two reasons. First, note that we are about to measure application's safety against OS command injection vularibility not its weakness. Second, appearantly the vector characterticity aspect of this triplet is not practical for our model. Given the above ,the triplet introduced by Gennari, J., and Garlan, D [14], is modifed to a linear combination of 1/(DER_m, DER_c and DER_i) values , as follows:

$$f(DER_m, DER_c, DER_i) = 1/(DER_m + DER_c + DER_i) \quad (12)$$

, where$DER_m$is damage/effort ratio of entry/exit points. These points return to methods which accept or process data that are originated outside of the system and quantify as follows:

$$DER_m = \sum_{i=0}^{M} ((a)_i/(b)_i) \quad (13)$$

, where*M:*Total number of entry/exit points, *a* is the level of privilege associated with $(\text{Entry/Exit point})_i$ and *b* isthe level of rights needed to access $(\text{Entry/Exit point})_i$.

DER_c is damage/effortRatio of Channels. Channels are the communication mechanisms used for system interaction with its environment, such as network or inter-process communication mechanisms. Channel damage/effort ratio is measured based on the restrictions imposed on the data that a channel can transmit via their protocols. Less restricted protocols ease compromising for attackers since they can transmit more types of data, such as executable codes. *DER* ratio for channels (DER_c) is evaluated in terms of number of data types that are restricted to transmit over channel's protocol as numerator over level of access right needed to access that channel. Hence, the larger numerator values show less restriction on data to transmit over that channel:

$$DER_c = \sum_{i=0}^{C} (a)_i/(b_i) \quad (14)$$

, where *C* is total number of channels, *a* is total number of data types that are restricted to transmit over $(\text{channel})_i$ and *b*represents the level of rights that is required to access the $(\text{channel})_i$

$DER_i$is damage/effortratio of untrusted data items. Untrusted data items are the external exited data stores that application uses them. DER for untrusted data items is measured based on restrictions areput on the data stores.

$$DER_i = \sum_{i=0}^{I} (a)_i/(b)_I \quad (15)$$

, where *I* is total number of external data stores that are utilized by application, *a*is total number of untrusted data items, and *b* is the level of rights needed to access that data items.

IV. Effectiveness Coefficient Numeric Value Attainment Method

In this section, we explain our approach for calculation of the effectiveness coefficient of each metric in mitigating certain vulnerability. Evidently, there are always a number of mitigation strategies that have been recommended to reduce the adverse effects of common vulnerabilities. However, they do not demonstrate comparable effectiveness. For example, suppose that "*Input Validation*" metric is far more effective than "*engaging Stored Procedures*" in improving safety against SQL Injection attacks. Therefore, we should consider a greater weight for "*Input Validation*". One of the common methods to find this parameter is to find a compelling argument from other reliable researches which in our case was not available. Therefore, we picked the alternative method of asking expert developers to fill out questionnaires while using the method.

To answer the question about "effectiveness extent of the preventaitve method" there are five options on the survey to choose from. We assign *eff* ∈ {0, 1, 2, 3, 4} from these options as follows:

1) *eff*=0 for "It has no effectiveness."
2) *eff=1* for "It has low effectiveness."
3) *eff=2* for "It is fairly effective."
4) *eff=3* for "I is highly effective."
5) *eff*=4 for "It is extremely highly effective."

Moreover, as all the respondents were not equally familiar with the subject, we also included a "familiarity weight" parameter as *fm*∈{0, 1, 2, 3, 4} in calculation of *"e"*. Similarly, we assign *fm* value from the survey options as follows:

1) *fm*=0 for "I have never heard about it before."
2) *fm*=1 for "I know this method, but never used it."
3) *fm*=2 for "I rarlely use this method."
4) *fm*=3 for "I ferequently use this method."
5) *fm*=4 for "I always use this method."

Another parameter that is involved in our calculation is respondent experience, We interprete value from their answer to correspondent question as follows:

1) *exp*=1 for "Below 1 year."
2) *exp*=2 for "Between 3 to 5 years."
3) *exp*=3 for "More than 5 years"

Moreover, we recruit fuzzy logic to transform the mentioned obtained rational values into numerical ones– $e \in [0, 1]$, and exploit them in our final safety calculation formula. In this formula , the product of respondents' "experience" and "familiarity" is considered as weight. The final value is the weighted average of effeciveness values. Then inorder to map the result to a number beween 0 to 1, we devided the result to 4 as the maximum value, which acures when all variables have their maximum value, and is calculated as follow.

m^* [Max(X)*Max(W)] / [m^* Max (W)]= m^* (12*4)/ (m*12) =4

Based on above discussion, the membership function for effectiveness of each metrics defined as follows:

$$(e)_{Metric\,n} = (\sum_{i=0}^{m}((Xn)i * (Zn)i * (Yn)i)/(\sum_{i=0}^{m}(Zn)i * (Yn)i)/4$$

$$\rightarrow (e)_{Metric\,n} = (\sum_{i=0}^{m}((Xn)i * (Wn)i)/\sum_{i=0}^{m}(W)i))/4 \quad (16)$$

, where X:*eff* $\in \{0, 4\}$, Z : *fm*$\in \{0, 4\}$, Y: *exp*$\in \{1, 3\}$, W: *exp*fm* $\in \{0, 12\}$, and m *:* number of respondents

We examined this approach with three respondents (m=3). Table 1 contains the generated "e" value for each metrics.

metric	"e" value
SQLInj-001	0.72
SQLInj-002	0.58
SQLInj-003	1.00
SQLInj-004	0.83
SQLInj-005	0.57
SQLInj-006	0.69
ShellInj-001	0.85
ShellInj-002	0.62
ShellInj-003	1.00
ShellInj-004	0.58
ShellInj-005	0.94
ShellInj-006	0.44

Table 1: *"e"* value for each metrics

V. RESULTS

To examine our model, various developers used the proposed metrics in their projects. The applications included different types of web applications and E-commerce/E-business applications. An Excel worksheet was made and presented to developers to enter the metric parameters to find the overall safety. Once each metric is calculated, the safety against SQL injection and Shell injection vulnerabilities can be found. Table 2 summarizes the detailed results for different tested applications.

Metric name	App1	App2	App3	App4	App5	App6	App7
SQLInj-001	3.00	2.00	4.43	2.75	3.00	1.44	4.00
SQLInj-002	1.00	1.00	0.79	1.25	1.00	0.73	0.93
SQLInj-003	0.00	2.00	0.71	0.00	0.00	2.20	3.00
SQLInj-004	1.00	1.92	1.86	1.25	1.35	1.93	1.30
SQLInj-005	0.00	14.08	8.67	1.00	1.20	9.29	5.67
SQLInj-006	0.00	0.01	0.00	0.25	0.40	0.37	0.04
ShellInj-001	1.00	4.00	1.00	1.00	1.25	4.60	1.23
ShellInj-002	1.00	0.00	0.00	1.50	1.00	0.30	0.02
ShellInj-003	0.00	2.00	0.71	0.00	0.28	2.20	0.83
ShellInj-004	1.00	1.92	1.86	1.25	1.50	2.50	2.87
ShellInj-005	0.57	3.56	1.09	0.60	0.32	4.50	1.09
ShellInj-006	0.13	0.06	0.13	0.25	0.38	0.04	1.23
OSAV	697.49	3439.82	1820.63	827.79	869.44	3978.27	2002.41

Table 2: Results of using metrics in different applications.

Furthermore, the examiners evaluated the usability and functionality of our formulas and metrics by means of another questionnaire. Tables 3 and 4 depict the results.

	Metricsare usable	Metrics increased safety	Definitely will use Metrics in future	Definitely will recommened Metrics to Colleagues	Easy to calculate formulas of metrics
Average score	89.00	68.00	74.71	60.43	94.43

Table 3: Average Results of usability questionnaires for formulas. The number are from 1 to 100.

	All metric's variables are necessary	Easy to find variables in application	No improvement required
Average score	82.29	80.14	64.71

Table 4: Average Results of funtionnality of metrics. The numbers are from 1-100.

VI. CONCLUSION

Using proper metrics in software engineering has not been very common as opposed to other engineering disciplines due to availability of such metrics. Furthermore, the need for safety and security metrics is probably the most important of all in-demand metrics in software engineering. This work is an attempt to fill out the lack of quantitative metrics in application development and software engineering. In this innovative method, new quantitative model for evaluating the safety of web applications is proposed. The metrics can quantify the overall safety of an application against known vulnerabilities. The main goal in developing this method was to provide an easy-to-use, scalable and flexible model for web application developers. In this way, they can measure the safety at different phases of development. This addresses the issue that web application security has to be looked at as an integrated factor in development and not as an add-on element. In addition to test of the method, different surveys were conducted to evaluate the usability of the formulas and metrics by developers. The feedback from web developers demonstrates that the proposed method is effective to provide a more secure application. This future work would enhance the experiments on the method in real application development.

VII. REFERENCES

1. CWE, "About CWE", http://cwe.mitre.org/about/index.html, n.p. , 2011, Last acessed on Sep. 25,2013.
2. Braungarten, R., "The SMPI model: A stepwise process model to facilitate software measurement process improvement along the measurement paradigms", 2007, PhD Thesis.University ofMagdeburg, Germany.
3. Brian, C., "Metrics that matter: Quantifying software security risk", Feb. 2006, Workshop on Software Security Assurance Tools, Techniques, and Metrics,. NIST Special Publication 500-265.
4. British Standard Inst., "Information Security Management. Specification for Information Security Management Systems (BS7799-2)", 1999, British Standard Institute, London.
5. British Standard Institute, Information Security Management. Code of Practice for Information Security Management.(BS7799-1)" , 1999, British Standard Inst., London.
6. CAPEC, "CAPEC-88: OS Command Injection", http://capec.mitre.org /data/definitions/88.html/ ,n.p. , June 21, 2013 , Last accessed on Sep. 25, 2013.
7. Microsoft, "Create Views", http://technet.microsoft.com/en-us/library/ ms175503.aspx ,n.p. , 2013 , Last accessed on Sep. 25, 2013.
8. CWE, "CWE/SANS Top 25 Most Dangerous Software Errors", http://cwe.mitre.org, n.p. , 2011, Web, Last acessed on Sep. 25, 2013.
9. CWE, "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')", http://cwe.mitre.org/data/definitions/78.html#Demonstrative%20Examples, n.p., 2011, Last accessed on Sep. 25, 2013.
10. Deborah R., Gangemi, G.T., "Computer Security Basics", chapter 3 :"Computer System Security and Access Controls", 1st Edition , July 1991, O'Reilly Media, ISBN 10:0-937175-71-4.
11. OWASP,"Defense Option 2:Stored Procedures", https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_2:_Stored_Procedures, , n.p., Dec. 6, 2012, Last accessed on Sep. 25, 2013.
12. Howard, M., "Fending off Future Attacks by Reducing Attack Surface",http://msdn.microsoft.com/en-us/library/ms972812.aspx, Feb. 4, 2003, Last accessed on Sep. 25, 2013.
13. McGraw,G., "Software Security: Building Security In", Feb. 2006, Addison-Wesley, ISBN: 0-321-35670-5.
14. Gennari, J., and Garlan, D., "Measuring attack surface in software architecture", 2011, Tech. Rep. CMU-ISR-11-121, Inst. for Software Research, School of Computer Science, Carnegie-Mellon University.
15. Microsoft, "How To: Protect From SQL Injection in ASP.NET",http://msdn.microsoft.com/en-us/library/ff648339.aspx, Last accessed on Sep. 25, 2013.
16. Goldberg, I., Wagner, D., Thomas, R., and Brewer, E., "A Secure Environment for untrusted Helper Applications (Confining the Wily Hacker)", July 1996, Proceedings of the 6th USENIX UNIX Security Symposium.
17. J. W. P. Manadhata. Measuring a system's attack surface. Technical Report CMU-CS-04-102, 2004
18. J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, AndyWigley,"http://msdn.microsoft.com/en-us/library/ff650175.aspx",Sep2005, Web, Access Date :Sep.25.2013
19. M.Howard, J.Pincus, J.M.Wing, "Measuring Relative Attack Surfaces", August 2003, Proc. Workshop Advanced Developments in Software and Systems Security
20. Manadhata, P. and Wing, J., " An Attack Surface Metric", Software Eng., IEEE Trans on ,Vol:37 , Issue: 3 , 07 June 2010,pages: 371 – 386.Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". Chapter 8: "Shell Metacharacters", 2006, 1st Edition. Addison Wesley, Page 425.
21. Mazinanian, D., Doroodchi, M. , Hassany, M., "WDMES: A Comprehensive Measurement System for Web Application Development" 2012, Telematics and Information Systems (EATIS), 6th Euro American Conf. on, pages:1 –8
22. Howard, M. and LeBlanc, D., "Writing Secure Code", Nov. 30, 2009, Microsoft Press, 2nd edition, ASIN: B0043M4ZPC
23. Howard, M., LeBlanc, D., and Viega, J."24 Deadly Sins of Software Security". "Sin 10: Command Injection." September 3, 2009 , McGraw-Hill, ISBN: 0071626751 , Page 171.
24. OWASP, "2010 OWASP Top 10", 2010.
25. National Vulnerability Database, "NVD Common Vulnerability Scoring System Support v2". National Institute of Standards and Technology. Last accessed on Sep. 25, 2013.
26. Manadhata, P. K. and Wing, J. M., "Measuring a System's Attack Surface," Jan. 2004, Technical Report CMU-CS-04-102, Carnegie Mellon Univ.
27. Roy, A. K. Singh, and A. S. Sairam, "Analyzing SQL Meta Characters and Preventing SQL Injection Attacks Using Meta Filter", 2011, Int'l Conf. on Information and Electronics Engineering, Singapore
28. S. R. Kumar, T, Alagarsamy K. "A Stake Holder Based Model for Software Security Metrics" ,2011, International Journal of Computer Science issues, Vol. 8, Issue 2, ISSN (Online):1694-0814, Available at: www.IJCSI.org
29. Jaquith, A., "Sample Questions for Finding Information Security Weaknesses", CSO, http://www.csoonline.com/article/221202/sample-questions-for-finding-information-security-weaknesses, May 18, 2007, Last accessed on Sep. 25, 2013.
30. SANS, "SANS Critical Vulnerability Analysis Archive", http:// www.sans.org, n.p. , March 16, 2007 , Last accessed on Sep. 25, 2013.
31. OWASP,"SecureCodingCheatSheet",https://www.owasp.org /index.php/Secure_Coding_Cheat_Sheet, n.p., April, 15, 2013, Last accessed on Sep. 25, 2013.
32. Cigital,"SecurityIssuesinPerlScripts",http://www.cgisecurity.com/lib/sips.html, Jordan Dimov, n.d. , Last accessed on Sep. 25, 2013.
33. OWASP,"SQLInjection Prevention Cheat Sheet", https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet, n.p., Dec. 2012, Last accessed on Sep. 25, 2013.
34. Microsoft,"SQLInjection",http://technet.microsoft.com/en-us/library/ms161953%28v=SQL.105%29.aspx, *n.d.*,Last accessed on Sep. 252013
35. "SQL injection",MS ISAC, http://msisac. cisecurity.org/ resources/reports /documents/SQLInjection.pdf , Stephanie Reetz , 23 January 2013, Last accessed on Sep. 25, 2013.
36. Litwin,P., Stop SQL Injection Attacks Before They Stop You",Microsoft,http://msdn.microsoft.com/en-us/magazine/cc163917.aspx, 2013, Last a, ccessed on Sep. 25, 2013.

37. Cisco,"UnderstandingSQLInjection",http://www.cisco.com/web/about/security/intelligence/sql_injection.html, n.p., n.d, Last accessed on Sep. 25, 2013.
38. Holm, H., Ekstedt, M., Sommestad, T., "Effort estimates on web application vulnerability discovery", 2013, 46th Hawaii International Conference on System Sciences.
39. Viega, J. and McGraw, G.,"Building Secure Software: How to Avoid Security Problems the Right Way",2002,Boston,Addison-Wesley
40. Martin B.,Brown M., Paller A., kriby D., Christey S., "2011 CWE/SANS Top 25 Most Dengrous Software Errors", 2011.