Journal of Computer Science

Original Research Paper

Securing Web Applications through a Framework of Source Code Analysis

[1]Alka Agrawal, [2]Mamdouh Alenezi, [1]Rajeev Kumar and [1]Raees Ahmad Khan

[1]Department of Information Technology, BBA University, Lucknow UP, India
[2]College of Computer and Information Sciences, Prince Sultan University, KSA, Saudi Arabia

Article history
Received: 18-09-2019
Revised: 20-09-2019
Accepted: 24-12-2019

Corresponding Authors:
Rajeev Kumar
Department of Information Technology, BBA University, Lucknow UP, India
Email: rs0414@gmail.com

Abstract: Source code analysis is becoming extremely important for the universal acceptance of web applications because the automated source code analysis tools play a key role in identifying and fixing security-related vulnerabilities. This paper proposes a framework for securing web applications through source code analysis. The framework has three prescriptive phases including executing and monitoring, classifying and controlling and refining and managing. The framework helps to examine the web application source code related to security issues. The executing and monitoring phase employs five different open source tools for statically analyzing the source code. According to the literature, there are nine broad categories of vulnerabilities in web applications. After filtration of these vulnerabilities, classifying and controlling phase categorize the vulnerabilities according to their severity level with the help of fuzzy analytical analysis process and suggestive measures. The refining and managing phase takes these measures and suggests changes to the source code to make it more secure. This framework was validated through a web-based hospital management system. The results of the validation showed that the framework implementation made the source code more robust towards the upcoming vulnerabilities and bugs.

Keywords: Web Application, Web Security, Security Vulnerability, Source Code, Static Analysis

Introduction

Web Applications, with the ubiquitous and ever-increasing usage, have become an inseparable part of our everyday lives. Consequently, the present context has witnessed that web applications are becoming more vulnerable because of the huge number of users associated with them (VanDen *et al*., 2018; Chess and McGraw, 2004). Further, hackers want to capture web applications to steal users' information. Unfortunately, most web applications are vulnerable to attackers due to the weakly designed and written source code. Designers must develop secure web applications by preventing vulnerabilities. Hackers can target security weaknesses in source code, which might be due to the web developers' lack of knowledge. In addition, most of the time, the developers follow bad coding practices to build web applications quickly. Hence, in the given scenario, web security related issues are increasing at an alarming pace. Source code analysis is one of the most significant actions to determine the vulnerabilities during the Web Application Development Life Cycle (WADLC) (Chess and McGraw, 2004). Further Static source code analysis is one of the most important activities to find bugs in the early stages of web application development.

There are many automated static source code analysis tools in the literature that could detect security vulnerabilities (Chess and McGraw, 2004; Standard, 1997). Source code analysis tools such as Arachni focus on the security of web application (Arachni, 2018). These automated tools were built to help developers remove security vulnerabilities at the early stages of WADLC. Before executing the codes, these tools scan the source code for potential security vulnerabilities in web applications including cross-site scripting and SQL injection. Unfortunately, relying only on automated scanning tools would lead to a large number of false positives. Despite being able to find bugs these analysis tools can also raise some false alarms. Some of the vulnerabilities are identified during the run time analysis including the complexity of the code, design flaws, etc. At present, the software industries are using most of the third party codes for agile and rapid software development. According to Positive Technologies Report, the medium level severity vulnerabilities have

increased from 97% to 100% in 2017 from the last year (Internet Security Threat Report by Semantics, 2016; Web Application Vulnerabilities: Statistics for 2017; 2018). This is happening because there are gaps between code analysis, updated vulnerability databases and developers reengineering process.

There is a compelling need for a mechanism for source code analysis to reduce the security related vulnerabilities from the beginning. Many practitioners are trying to develop a mechanism for producing secure source code. In this work, we are proposing a common framework for producing secure code through static source code analysis. The framework poses three phases including *Execute and Monitor*, *Classify and Control* and *Refine and Manage*. The source code of a web-based hospital management system is adopted to empirically validate the proposed framework. We chose a hospital system since it has very sensitive as well as classified personal and medical information such as blood reports, treatment records of the patients, etc. During the implementation of phase 1, detection of security vulnerabilities is done in source codes of web applications through five different open scanning tools including Arachni (2018), FindBugs (2015), SonarLint (2017), EasyPMD (2015) and JArchitect (2018). Phase 2 categorizes the vulnerabilities and calculates their priorities through fuzzy analytical hierarchy process because the prioritization of vulnerabilities is a multi-criteria decision problem. After that, according to the defined severity of vulnerabilities, suggestive measures are presented. Phase 3 refines the process of managing web security for developers.

Literature Review

Source code analysis can be used to guarantee at least safe and secure delivery of web application to consumers (Ahmed and Ullah, 2018). Vulnerability and bug discovery seems to be a serious problem in security assurance policy. Code analysis is an effective way to secure web applications through security vulnerabilities and flaws identification. Further, run time code analysis (dynamic analysis) is very costly and time-consuming rather than static code analysis. In previous years, plenty of work has been done to achieve Web Application Security by different frameworks, methodologies and tools. Some of the pertinent initiatives are shown in Table 1.

Table 1: Summery of the pertinent works

Sr. No.	Reference	Summary of the Contributions
1.	Verma and Sharma (2019)	This work reviews three important and most commonly used static analysis tools CppCheck, FlawFinder and Visual Code Grepper (VCG). The author discusses the features, importance and limitations of each tool. The conceptual and empirical comparison has also been done. The authors found VCG to be the best tool amongst all the three.
2.	Smith *et al*. (2018)	Authors presented exploratory research on how developers implement static analysis tools. For this work, they took ten developers and provided them with FindBugs tool to implement it on the source code. Results were positive as they found the static analysis tools helpful for developers. The authors suggested that analysis tools should help with preliminary information about the tool as well, rather than only providing a search of relevant vulnerabilities.
3.	Nunes *et al*. (2018)	This work provided a benchmark for differently used static analysis tools based on PHP language on four different projects scenarios of web applications. The author discussed that different results are gained by analyzing the same code snippet with different static analysis tools. Hence, a common benchmark will be able to set standards for the developers to enlist.
4.	Zampetti *et al*. (2017)	Authors studied the use of static analysis tools in 20 Java open source projects acquired from GitHub and using Continuous Integration infrastructure. The paper investigated which are the tools used for Continuous Integration. The results analyzed that most of the vulnerability issues are license based problems or coding adherence rules.
5.	Beller *et al*. (2016)	This work discussed problems that arise in using Automated Static Analysis Tools for both the dynamically typed languages and strict languages. Findings of this work specify that open source software developers need to be more aware of the static analysis tools usage and its behavior. Practical guidelines for the users are also created in this work which is further useful for researchers as well as practitioners.
6.	Perl *et al*. (2015)	This work proposed a new analysis tool VCC to find the flaws during static analysis and flag these so that special attention is drawn to these flaws. Authors combine code metric analysis method and metadata available open source to come up with a new method of analysis. This method or tool also assures false alarm rate reduction to 99 percent. Further, they validate this tool by comparing it to another tool FlawFinder.
7.	Yamaguchi *et al*. (2014)	Authors provided a novel data structure to represent source code so that finding and debugging of vulnerabilities might become easier and time efficient. This novel data structure is a code property graph, which is a combination of Abstract Syntax Tree, Control Flow Graph and Program dependence graphs. This data structure was able to find only some kind of common vulnerabilities that usually occurs in source codes. Also, this study may not be helpful with the futuristic web applications, which have thousands of lines of code.
8.	Kulenovic and Donko (2014)	Authors provided a critical review of static code analysis methods and how useful are these methods in finding vulnerabilities in source code. This paper strengthens the fact that static code analysis is the most powerful method to find and debug vulnerabilities in source codes. Also, the author believes that algorithms for static analysis are improving day by day.
9.	Meneely *et al*. (2013)	Authors found 68 vulnerabilities in a very well-known Apache HTTP server. 124 vulnerability contributing commits were found when manually scanned by authors. Authors further investigated the vulnerability commits and provided guidelines for developers to know the reason behind the generation of vulnerabilities in source codes.
10.	Heckman and Williams (2011)	This project studied nine tools including SATABS and some important commercial tools. Author applied tools against SAMATE referenced datasets. Methodology used is repeatable for all tools. The results obtained empirical evidences that support popular propositions. At the end authors provides recommendations for improving the reliability and usefulness of static analysis tools.

By the relevant work presented in Table 1, we can conclude that there is plenty of work that has been done for the security of web applications in the previous years. Some important and popular tools such as FindBugs and Visual Code Grepper (Visual Code Grepper- Code Security Scanning Tool, 2016) have also been proposed which are working on very critical vulnerabilities such as Cross-Site Scripting and SQL injection. However, every method and tool would entail pros and cons (Heckman and Williams, 2011). These tools/methods do not provide a complete setup for identifying as well as removing and managing the vulnerabilities. It has been critically observed that none of these proposed tools or methodology can be presented by the industry as a complete package for delivering secure source code. Hence, there is a need for a common framework, which provides a proper and complete setup for identification, prioritization and removal of high priority-based vulnerabilities. The proposed framework fills the gap between technical and theoretical paths of vulnerability identification and removal.

Proposed Framework

Web applications development organizations are always focusing on new ideas to gain the trust of the users. In addition, organizations wish to secure web applications, which provide longer services to increase user satisfaction (Nausheen and Begum, 2018). The source code is very helpful for an organization to construct a secure web application. The mistakes that developers make at the code level and configuration level are mitigated at the time of static source code analysis. Further, source code analysis process identifies security vulnerabilities and verifies if the key security controls are implemented.

Producing a secure source code is a crucial task for practitioners. To fill the gap between developers and secure code, there is a need to integrate the whole process for scanning, detecting, mitigating the security vulnerabilities and flaws during source code analysis. Moreover, the integration process will also reduce the cost and rework involved otherwise. Finding ways to produce secure source code is still a challenging task (Heckman and Williams, 2011). Keeping the need and significance in mind, authors have structured a hierarchical description of proposed framework including premises, generic guidelines and framework development process to be followed. Premises and generic guidelines talk about the planning or training to be done to develop framework for any specific case. Framework development is further divided into three major parts: Execute and Monitor the Source Code Analysis, Classify and Control the Security Vulnerabilities and Refine and Manage the Procedure. The description of the proposed framework with premises and guidelines are in order as enunciated below:

Framework Development

The motive of the secure source code analysis process is that the software behaviour would be fully operative under hostile conditions. Analysis of source code infers analysis to find out and list the number of vulnerabilities and bugs to develop a more secure web application from the initial phase of code review. There are different source codes for different languages. Software developers mostly use the already written source codes which are also more vulnerable. The ultimate objectives of source code analysis are to identify and mitigate the security vulnerabilities and flaws before executing and dynamically analyzing the code.

At present, developers are trying to focus on security during source code analysis through the automated tools (Larrucea *et al*., 2019). Unfortunately, these automated tools are available only for limited languages and the reliability of these tools account for only 40% (Arachni, 2018). An effective source code analysis process should check the entire steps, rather than just doing the analysis. A process of source code analysis is needed to ensure that the source code can protect its assets from attacks. An appropriate and accurate secure source code analysis activity if implemented would make the software more profitable (FindBugs, 2015; Sonarlint, 2017). In addition to this, an effective and prescriptive process of secure source code analysis, specifying very clear prioritized activities, may be advantageous in different perspectives.

In order to gain insight into the quality of web application, a unified process for secure source code analysis is developed to spot the security vulnerabilities and flaws during the analysis process and to mitigate the same. After the coding premises and the guidelines development, a combination of source code analysis during web application development will reduce the cost of damages and risks associated (EasyPMD, 2015). Step by step, a three level strategy for secure source code analysis is been established as a framework which has been defined hierarchically in coming phases:

Execute and Monitor the Source Code Analysis

In this phase, source code should be executed through a common analyzer in a reasonable order including data flow analysis, semantic analysis, control flow analysis and configuration analysis. Further, during data flow analysis, the analyzer detects the flow of malicious data (Huang *et al*., 2004). During semantic analysis, analyzer searches for vulnerable functions used in the code (Detection of Vulnerabilities in Programs with the Help of Code Analyzers, 2008). During control flow analysis, the analyzer tracks the sequence of operations to detect improper coding constructs (Code Optimization Control Flow Graph, 2018). During configuration analysis, analyzer parses and analyzes the application deployment/ environment settings in configuration files (Paladion, Source Code Analysis Suite, 2018). The analyzer scans the source code and identifies the vulnerability and flaws. The process should be monitored and identified vulnerabilities and flaws should be verified through the practitioners. Identified blacklist code and whitelist code should be documented. Found vulnerabilities and flaws code is sent to

DOI: 10.3844/jcssp.2019.1780.1794

repository 1 and if not it is sent to repository 2. From repository 1 code is sent for the step 2 of framework. A prescriptive step in executing and monitoring the secure source code analysis is shown in Fig. 1.

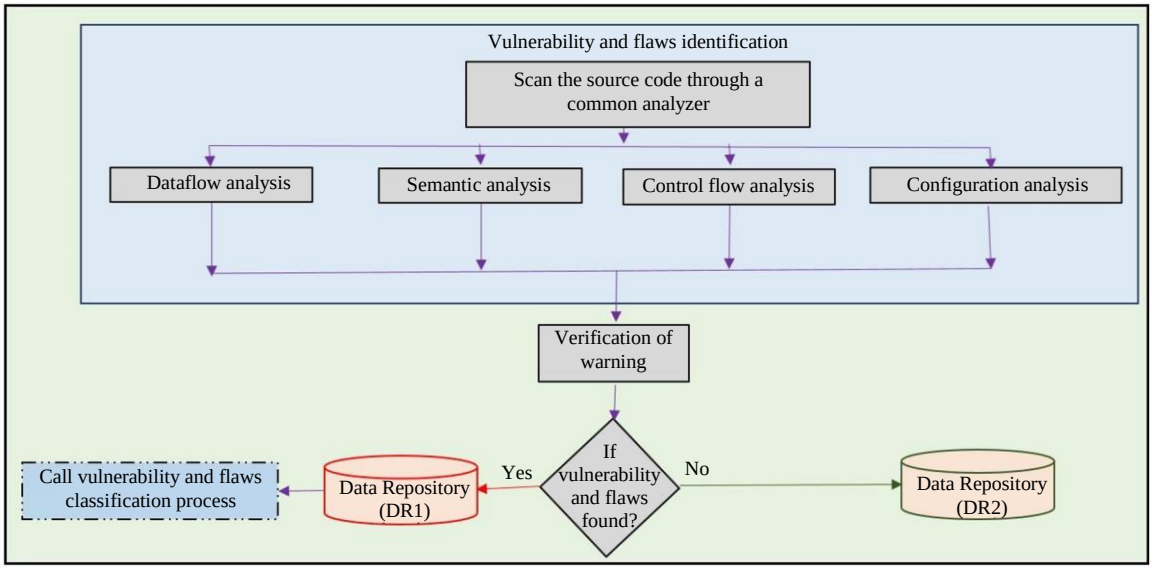

Fig. 1: Prescriptive steps for executing and monitoring the secure source code analysis

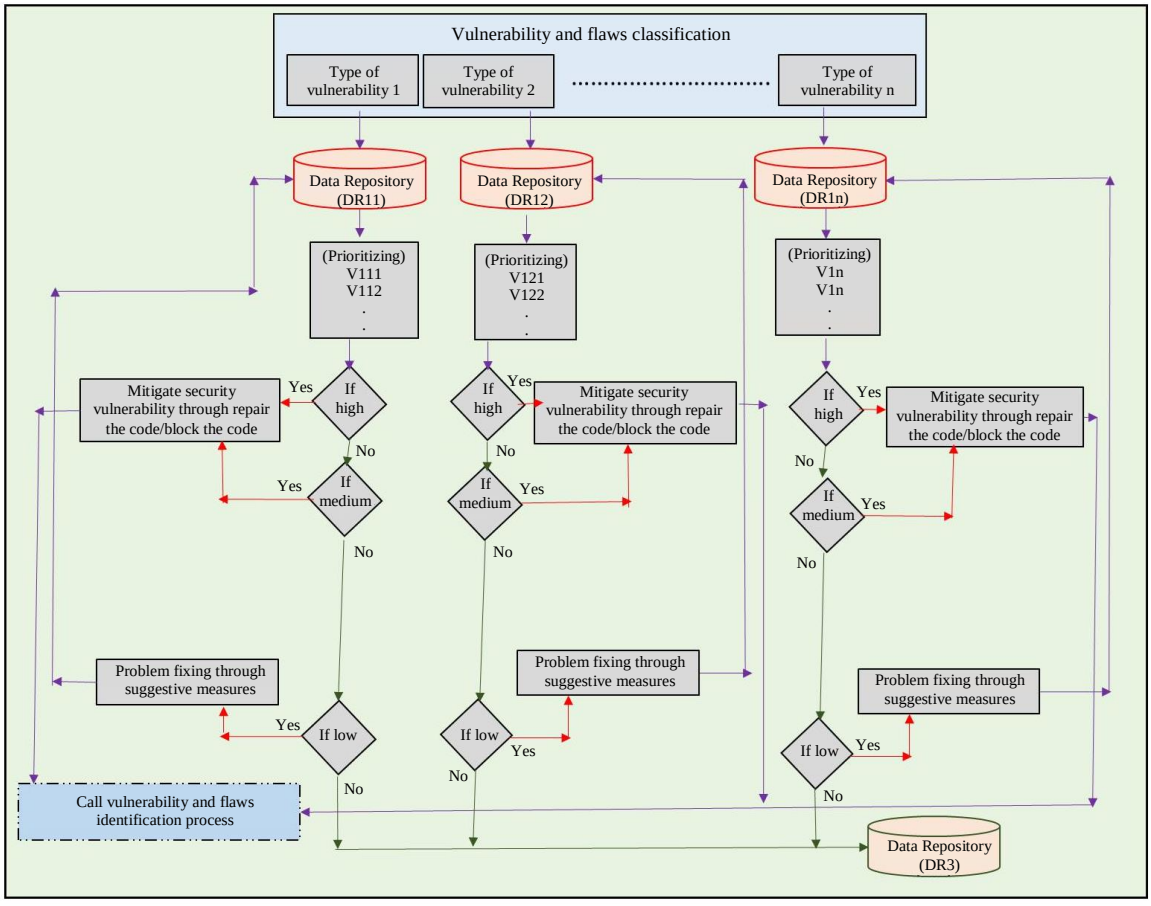

Fig. 2: Prescriptive steps for classifying and controlling the secure source code analysis

Classify and Control the Security Vulnerabilities

After successfully implementing phase 1, the identified security vulnerabilities and flaws should be classified into different categories including SQL injection and Cross-site scripting, etc. In addition, prioritize the vulnerabilities and flaws according to their severity levels to reduce the cost and time during the mitigation plan. Also, the severity levels should be classified into three levels including high, medium and low. Repair the code or block the code to mitigate the high level and medium level of security vulnerabilities. Problem shall be fixed through suggestive measures to mitigate the low level of security vulnerabilities. A summary report of the analysis should be prepared to finally summarize the actions associated with the source code. A prescriptive step in classifying and controlling the secure source code analysis process is shown in Fig. 2.

Refine and Manage the Procedure

After successfully implementing phase 2 of the analysis process, all the repositories of source code should be merged into a single repository. Again, source code analysis should be analyzed by manual analysis. Identified logical errors and flaws should be mitigated through suggestive measures. Further, coding guidelines should be refined and prioritized and finally facilitating the codes into software development life cycle. A prescriptive step in refining and managing the secure source code analysis process is shown in Fig. 3.

In the present scenario, dependency on the web application is so high that life cannot be imagined without them. With the overall advantages of web application and the security design on them, there is also a quantum of fear as well. Fear of being insecure and the looming threat of being hacked is always there besides the other apprehensions that come with the dependency on web application (Hussain *et al*., 2018). Thus, the consideration for web security during source code analysis emerges as a helpful solution for the developers as well as the users. The framework aims at preventing security problems by building a web application without security holes (Beller *et al*., 2016 and Yamaguchi *et al*., 2014). Phase wise implantation of the framework is done in the next section of this study.

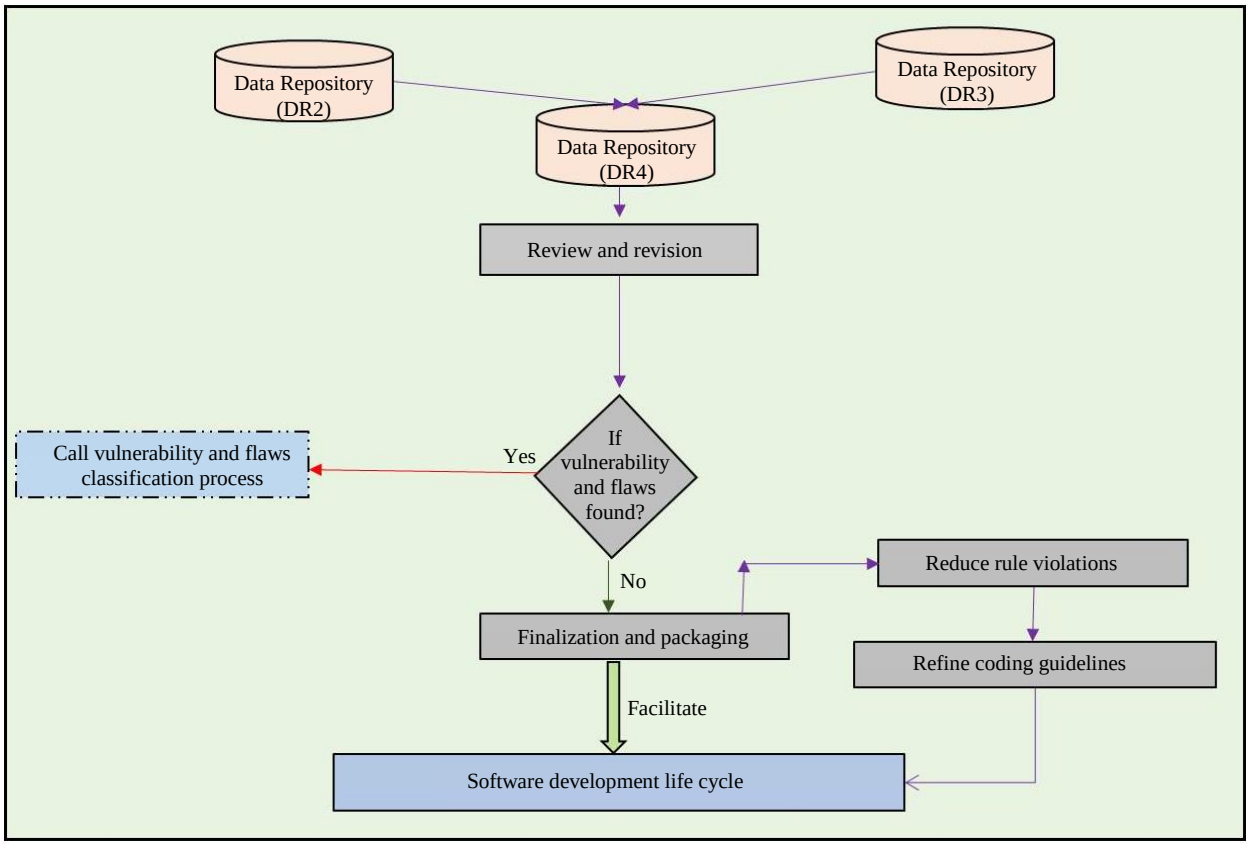

Fig. 3: Prescriptive steps for refining and managing the secure source code analysis

Implementation of the Framework

Due to the wide applicability of information systems, web security has become a crucial component during web application development. Indeed, web application faces threats from various potential malicious adversaries that are rising every day (Huang *et al*., 2004). These threats can impose a vast challenge to developers in planning measures as a portion of their risk management activities as well as in designing the appropriate security requirements and policies. This is due to the degree of subjectivity in how security is being perceived and subject to different levels of concerns. Moreover, numerous web applications are developed without paying due attention to security issues including the SQL injection, cross-site scripting and bad practices of codes (Ayeni *et al*., 2018). Further, source code analysis is one of the most significant features for securing the web application that calls for high attention amongst the engineers. To identify and mitigate security vulnerabilities during source code analysis, this paper has taken the open source code of hospital management system (Hospital Management System in Java Using NetBeans with Source Code, 2018). Due to very sensitive information of the patients, effective source code analysis is essential for securing the web application. In the project, 23 main classes have 7046 lines of codes. For identifying the vulnerable codes, phase 1 of the framework is implemented as follows:

For creating the challenges in the security of web application, vulnerable codes are responsible (Gürses and Santen, 2006). To prevent flaws and reducing the testing efforts, producing effective source code is an important but crucial task. During the implantation of phase 1, this paper uses the five open source code analysis tools including Arachni, FindBugs, SonarLint, EasyPMD and JArchitect. The results are obtained by powerful web application scanner tool designed for web languages such as Java, Javascript, AJAX, HTML5, etc. Arachni is smart to train itself by monitoring and learning from the web application's behavior. This framework provides great coverage to modern web applications due to its integrated browser environment (Arachni, 2018). The source code was analyzed online through Arachni and 135 vulnerabilities were found. According to the results, 121 vulnerabilities were detected in the low level, 8 vulnerabilities were found in the medium level and 6 informational vulnerabilities were traced.

FindBugs is an open source static code analyzer for JAVA, which was released in the year 2006. It is available both in the command line cloud and GUI (FindBugs, 2015). It is used as a plugin in Netbeans IDE 8.2. The source code is analyzed on FindBugs and 99 vulnerabilities are found and categorized into categories including bad practice, correctness, experimental, internationalization, malicious code vulnerability, multithreaded correctness, performance, security and dodgy code. After scanning the code, FindBugs ranks the bugs in four severity levels which are: Scariest, scary, troubling and of concern. SonarLint scanner scans the code that gives instant feedback as the code is written by the coder. It supports C#, VB.NET and Java languages in different IDEs such as Eclipse, Visual Studio and Atom. SonarLint is more of a spellchecker kind of tool that can store some quality rules. It alerts the developer while writing the code in case of any rule violation that might occur. The source code was analyzed through SonarLint and 100 vulnerabilities were found.

EasyPMD scanner scans the code that collates the results of scanning source code. It is an extension tool of PMD tool plugin with NetBeans 8.0. Further, PMD is a Java library tool that scans the Java code for possible violations of the already written rules along with the user's repository of written rules. It is available in the library of Java application and as a separate application. The source code was analyzed through EasyPMD and 92 vulnerabilities were detected. JArchitect scanner shows the results of scanning source code. It is a static analysis tool for Java source code. It analyses the code for certain defined quality standards and rules and presents vulnerabilities or bugs using dependency graph and dependency matrix. It is often called as a Swiss army knife for the Java developers. Reputed software development organizations such as Samsung and IBM use it for analyzing the source codes. The tool helped to detect the vulnerable line of code. The source code was analyzed through JArchitect and 103 vulnerabilities were found (JArchitect, 2018). These codes were not giving compilation errors but still vulnerabilities and rules violation were there. These five tools found a different number of vulnerabilities and flaws (Pistoia *et al*., 2007) that are enumerated in Table 2.

Table 2 enlists the problems in the source code of the hospital management system that could be found through a common analyzer doing data flow analysis, control flow analysis, semantic analysis and configuration analysis. The analyzer found four types of issues including: Potential SQL injection, the class contains a public variable, operation on primitive data type and public class not declared as final. According to the phase 2 of the framework, authors categorized the security vulnerabilities and flaws into seven categories for a web application. The definitions of security vulnerabilities are shown in Table 3. Table 3 highlights the vulnerabilities which are related to web security. Every web application has different usage in business, environment and purpose. Potential risks and threats should be defined with regards to the protected values and the weaknesses arranged accordingly (Abomhara, 2015). After verifying the alarms, Table 4 enumerates the actually identified vulnerabilities that may be exploitable.

Table 2: Problems found by using different tools

Categories of analysis	Rules	Arachni (2018)	FindBugs (2015)	SonarLint (2017)	EasyPMD (2015)	JArchitect (2018)
Data flow analysis	Privacy violation	X	X	X		X
	Integer over flow		X			
	Path manipulation	X		X		X
	System information leak		X		X	X
	Setting manipulation	X	X		X	
	String termination error	X				X
	Resource injection		X			
	Illegal pointer value			X		X
	Out-of-bounds	X				
Control flow analysis	Null dereference		X		X	
	Missing check against null			X		X
	Use after free	X		X		
	Redundant null check		X		X	X
	Insecure temporary file		X	X		X
	Uninitialized variable	X		X	X	
	Double free memory leak		X	X		X
	Unreleased resource race condition	X	X			
Semantic analysis	Insecure randomness			X	X	X
	Heap Inspection	X		X		
	Command injection	X	X	X	X	X
	Process control	X		X		X
	Portability flaw		X			
	Format string	X		X	X	X
	Cryptographic hash	X			X	
	Insecure compiler optimization	X	X		X	X
	Unchecked return value	X		X		X
	Often misused	X	X			
	Dangerous function	X	X	X	X	X
Configuration analysis	Dead Code	X	X	X	X	
	Password management			X		X
	Code correctness	X	X	X	X	X
	Type mismatch		X		X	
	Poor style	X	X	X	X	X

Table 3: Classification of security vulnerabilities

S.N.	Web security vulnerability	Description	References
1.	Common directory	Most of the web applications are built using common files and directories, that's why hackers focus on accessing these common directories by sending requests with most known names. This may lead to the sensitive database which the web application is using. This problem is known as common directory reference vulnerability.	Common Directories Detection, 2017
2.	Missing 'StrictTransportSecurity' Header	For security reasons, web application developers use HTTP Strict Transport Security (HSTS) to follow encryption standards. Cybercriminals attempt to get sensitive information which is passed from client to server by using HTTP instead of HSTS. This kind of vulnerability is called Missing Strict Transport Security Header.	Missing 'Strict-Transport-Security' Header, 2017
3.	Unvalidated Redirect	An unvalidated direct occurs when web application site redirects to another malicious site to modify the parameter value. Javascript is mostly used to redirect a browser to an arbitrary URL.	Unvalidated Redirection, 2018
4.	Common Sensitive File	It happens sometimes that some files got unused by time but are not removed by the administrator or forgotten. These specific files are weak points and vulnerable to hackers easily. This vulnerability is called the problem of the common sensitive file.	Arachni Common File, 2018
5.	Password field with Autocomplete	To improve the usability of web page, developers usually give auto-complete on password as well as on user id. Although it improves usability but it also increases the chances of attacks by hackers who visit that page.	The Autocomplete Attribute and Web Documents using XHTML, 2011
6.	Missing 'Xframe-Options' Header	HTTP response header can be used to indicate whether or not a browser should be allowed to render a page in a <frame>. When X Frame options are missing in a web application, it may lead to clickjacking by hackers.	Strict-Transport-Security' Header, 2018
7.	Cookie Set for Parent Domain/Insecure Cookie/HttpOnly Cookie	HTTP by itself is a stateless protocol. By the use of HTTP cookies, one can differentiate between the authentic and unauthenticated user. This further lessens the chances of hacking. Hence, the usage of insecure cookie lets the unauthenticated person access the sensitive information from the web application.	Arachni Insecure Cookies, 2018

In Table 4 total vulnerabilities found are 515 (128+95+97+92+103) which are further divided as true positive, true negative, false positive and false negative. In this work we are focusing on true positive vulnerabilities because true positive is a successful identification of the attack. And Table 4 has 66 vulnerabilities found as true positive in all which includes the common directory, missing 'strict-transport-security' header, unvalidated redirect, common sensitive file, password field with auto-complete, missing 'x-frame-options' header, a cookie set for parent domain/ insecure cookie/ HttpOnly cookie. To measure the severity of the web security vulnerabilities, this paper uses the fuzzy Analytical Analysis Process (AHP) because prioritization of vulnerabilities is a multi-criteria decision-making problem. To evaluate the severity of security vulnerabilities of web applications, AHP is one of the most important methods (Lokhande and Meshram, 2016). Also, it facilitates apt decisions among the multiple conflicting criteria and decisions (Mu and Pereyra-Rojas, 2017). In daily life, multiple criteria problems can be solved using AHP such as a selection of one criterion from different criteria (Mardani *et al*., 2015). The stepwise process to measure the severity of security vulnerabilities has been given in Fig. 4.

During the assessment of number of vulnerabilities in the web application, source code is identified. In the next step, information about the identified vulnerabilities is gathered and a questionnaire to collect the priorities from the security experts of web application security is prepared. Next, a hierarchy of these issues/vulnerabilities is created. This is followed by the step to prepare a pair-wise comparison matrix that helps a person in making the decision easier. The input proposes pair-wise comparisons to produce the judgment matrix. Saaty (1985) proposed pair-wise comparisons to create the judgment matrix that is used in the AHP technique. Corresponding linguistic scale for membership functions lies between 1 and 9. After constructing a pair-wise matrix of expert input, Consistency Ratio (CR) is calculated to control the results of the AHP method (Mu and Pereyra-Rojas, 2017). If the CR is less than 0.1, then the weight of each input is calculated. If the CR is greater than or equal to 0 then the refined pair-wise matrices are prepared and the process is repeated again. Further, after aggregating the pair-wise comparison matrix, CR is calculated and verified again. Table 5 shows the aggregated pair-wise comparison matrix for the issues of web security.

For defuzzification, an alpha cut method is used (Dymova *et al*., 2015). The next step is to determine the eigenvalue and eigenvector of the pairwise comparison matrix. The purpose of calculating the eigenvector is to determine the aggregated weight of a particular criterion. The aggregated results in terms of weights are shown in Table 6.

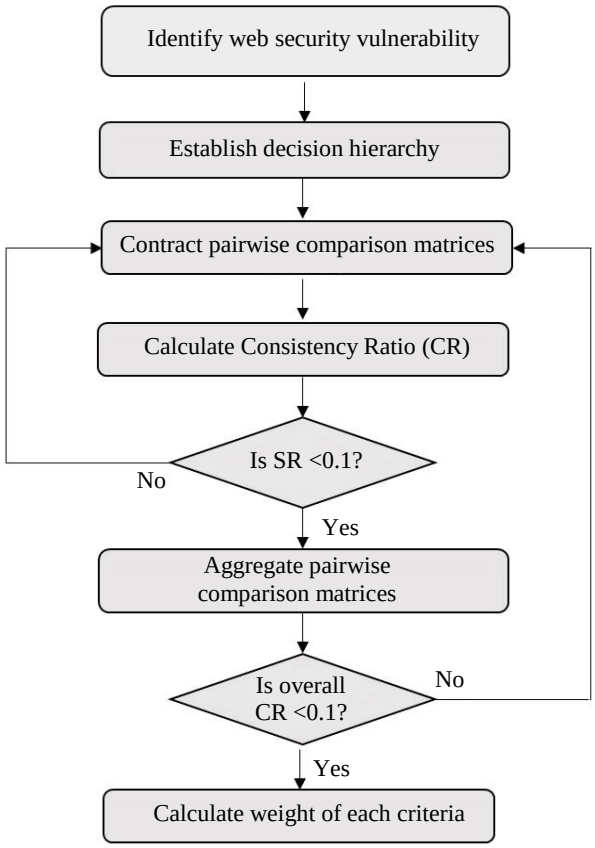

Fig. 4: Stepwise process for prioritizing the security vulnerabilities

Table 4: Identification and verifying the security vulnerabilities

S.N.	Web Security related Issues/Tools	Symbol	Arachni	FindBugs	SonarLint	EasyPMD	JArchitect		True positive
1	Common Directory	CD	119	87	82	80	95	Verify the	57
2	Missing 'StrictTransport-Security' Header	STS	1	1	2	2	1	Warnings	1
3	Unvalidated Redirect	UR	1	2	1	1	2	(False	1
4	Common Sensitive File	CSF	2	1	3	2	1	Positive)	2
5	Password field with Auto-complete	PFA	1	1	2	2	1		1
6	Missing 'X-frame-Options' Header	XOH	1	1	3	2	1		1
7	Cookie Set for Parent Domain/ Insecure Cookie/HttpOnly Cookie	CDH	3	3	4	3	3		3
Total Issues Found			128	95	97	92	103		66

Table 5: Aggregated fuzzify pair-wise comparison matrix

	CD	STS	UR	CSF	PFA	XOH	CDH
CD	1,1,1	1.0000, 1.5157, 1.9331	0.4896, 0.6372, 1	0.4152, 0.5743, 1	0.2215, 0.2871, 0.4152	0.3146, 0.4610, 0.8705	0.6575, 1.1653, 1.6883
STS	-	1,1,1	0.5743, 0.6657, 0.8022	0.3039, 0.3936, 0.5661	0.2679, 0.3521, 0.5176	0.1663, 0.1969, 0.2531	0.3930, 0.5743, 1.0564
UR	-	-	1,1,1	1.0000, 1.3195, 1.5518	0.3009, 0.4352, 0.8027	0.8027, 0.8705, 1	1.2619, 1.8250, 2.4334
CSF	-	-	-	1,1,1	0.5386, 0.9143, 1.5836	0.6083, 1.0592, 1.6829	0.7503, 1.3465, 1.9611
PFA	-	-	-	-	1,1,1	0.4152, 0.6372, 1.1791	0.9465, 1.1095, 1.2457
XOH	-	-	-	-	-	1,1,1	1.8881, 2.5508, 3.1697
CDH	-	-	-	-	-	-	1,1,1

Table 6: Weights and ranks of vulnerabilities

	CD	STS	UR	CSF	PFA	XOH	CDH	Weights	Priority	Severity
CD	1	1.4912	0.6910	0.6410	0.3027	0.5268	1.1691	0.1821	1	High
STS	0.6706	1	0.6770	0.4143	0.3724	0.2033	0.6495	0.1681	2	High
UR	1.4470	1.4771	1	1.2977	0.4935	0.8520	1.8364	0.1571	3	Medium
CSF	1.5600	2.4137	0.7706	1	0.9636	1.1024	1.3511	0.1484	5	Medium
PFA	3.3036	2.6853	2.0263	1.0378	1	0.7172	1.1028	0.1689	4	Medium
XOH	1.8982	4.9188	1.1737	0.9071	1.3943	1	2.3852	0.0789	7	Low
CDH	0.8554	1.5397	0.5445	0.7401	0.9068	0.4193	1	0.0965	6	Low

The weights obtained determine the priorities and severity. Priority with numbers 1 and 2 is considered as high severity problems. These vulnerabilities should be solved immediately. Priority with 3, 4 and 5 should be solved after solving the high-level severity vulnerabilities. Low-level severity has the lowest priority which is Missing ‘X-frame-Options’ Header and Cookie Set for Parent Domain/Insecure Cookie/HttpOnly Cookie. These low-level vulnerabilities should also be solved but after the higher and medium level vulnerabilities are solved. The static analysis focuses on solving as many vulnerable holes as possible before delivering the web application to the end user.

Web security has multiple issues in form of vulnerabilities in the source code that should be carefully assessed to get the secure web application. Vulnerabilities prioritization seems to have different types of criteria within it. For instance, to assess different types of vulnerabilities, one needs to assess its issues including common directory, SQL injection, etc. Important tasks for mitigating the issues according to ranks are discussed. After the identification and prioritization of vulnerabilities/issues, the next step is to mitigate the issues according to its priority. Every automated tool has a vulnerability database and coding rules. The common directory has found 57 vulnerabilities. Issues are discovered including class contains a public variable, operation on primitive data type and public class not declared as final. A number of vulnerabilities are repetitive in each class and mitigation of these issues are as follows:

The web application appears to allow SQL injection via a pre-prepared dynamic SQL statement. No validator plug-ins was located in the application's XML files. For mitigating the issues of public class, the class is not declared as final as per OWASP bunch of best practices. It has no classes which are inherited from the final class. The classes which are not declared as final may allow an attacker to add malicious classes into it. This can be resolved by manually inspecting the code to determine whether or not it is practical to make this class final. For mitigating the issues of operation data type, the code appears to be carrying out a mathematical operation on a primitive data type. In some circumstances, this can result in overflow and unexpected behavior. The solution is to check only the code snippet manually to determine the severity of the problem. For mitigating the issues of the class contains a public variable, the class variable should not be called or accessed using get or set methods. It is considered unsafe to have public fields or methods in a class unless required. This is so because any method, field, or class that is not private is a

potential opportunity of attack. Further, common directory traversal, also known as path traversal, is ranked number 13 on the CWE/SANS Top 25 Most Dangerous Software Errors. An example of common directory vulnerabilities is shown in following code snippet.

```
public class About {
   public static void main(String[] args) {
      File file=new File(args[0]);
   }

}
```

Further above code is found in About Class of the open source code of the hospital management system. Further, the common directory vulnerabilities will fortify and flag the code even if the path/file doesn't come from user input like a property file. The best way to handle these is to normalize the path for user data input first and then validate it against a list of allowed paths. To reduce the problem, following shows the code snippet.

```
public class About {
   public static void main(String[] args) {
      File file=new File(args[0]);
      if (!isInSecureDir(file)) {
         throw new IllegalArgumentException();
        }
        String canonicalPath = file.getCanonicalPath();
      if (!canonicalPath.equals("/img/java/file1.txt") &&
        !canonicalPath.equals("/img/java/file2.txt")) {
        // Invalid file; handle error
   }

   FileInputStream fis = new FileInputStream(f);
}
```

Similarly, authors tried to reduce the other issues of the common directory including modifications in the codes and fixing the issues. Another high severe vulnerability is found during scanning of the codes, i.e., missing HSTS. When someone doesn't use HSTS in accessing the website or uses only HTTP protocol to access, they are more vulnerable to be attacked by Man in the Middle Attacks. This type of attack is called the HTTP Strict Transport Security Vulnerability. Use of HSTS protocol reduces the possibility of the occurrence of a Man in the Middle attack. Further, Unvalidated Redirects occur when the user is directed to phishing or malicious site by accessing the site due to which the user's credential information can be hacked. Although this vulnerability is unavoidable, still it can be lessened with no involvement of user parameters in redirection. An example of unvalidated redirects is shown in following code.

```
if (domain != null && !domain.isEmpty()) {
 response.sendRedirect("https://" + domain +
request.getServletPath() + "?" +
request.getQueryString() + "&markAs=true");
}
```

To reduce the problem, shows the solution in following code snippet.

```
response.sendRedirect("https://" + domain +
getUrl(request) + "&abredir=true");

private String getUrl(HttpServletRequest request) {
return request.getServletPath() + "?" +
request.getQueryString();
}
```

Another high severe vulnerability is found while scanning the codes, i.e., Password field with Auto-complete. Today's browsers such as Mozilla and Chrome store the credentials entered by the user in HTML forms. This function can save the information of the user on the local computer and it can be used maliciously by another unauthorized person. Further, an attacker who finds a separate application vulnerability such as cross-site scripting may be able to exploit this to retrieve a user's browser-stored credentials. An example of a Password field with Autocomplete is shown in following code.

```
jLabel2.setText("Password");
jButton1.setText("OK");
jButton1.addActionListener(new
java.awt.event.ActionListener() {
   public void
   actionPerformed(java.awt.event.ActionEvent
   evt) {
      jButton1ActionPerformed(evt);
   }
});
```

In order to address the problem identified, the authors created a temporary text box above the password field and hide it in the following manner:

```
<label>Password:</label>
<input type="text" style="display:none;">
```

It will make the username as text field to not show any previously typed words in a drop down. Since there is no attribute like name, id for the input field <input type="text" style="display:none;">, it wouldn't send any extra parameters also. Further, common sensitive files cannot be logged as they may contain sensitive information such as the username and password. Hence, this problem should be analyzed and solutions should be provided to confront this situation. To avoid this, file

should be restricted for access or should be removed from the website. If the above-mentioned solution doesn't work, then the solution is to check the whole code manually. This would protect any sensitive information from being passed on to the other server and user.

Missing 'X-frame-Options' Header risks the attack of ClickJacking. ClickJacking is taking the user to another malicious website or database by creating different keystrokes or by trapping it through clicking some Headers which are made using JavaScript. An example of missing X Frame Options header is shown in following code snippet.

Web.xml

```
<filter>
 <filter-name>UrlRewriteFilter</filter-name>
 <filter-
 class>org.tuckey.web.filters.urlrewrite.UrlRewrite
 Filter</filter-class>
</filter>

<filter-mapping>
 <filter-name>UrlRewriteFilter</filter-name>
 <url-pattern>/*</url-pattern>
 <dispatcher>REQUEST</dispatcher>
 <dispatcher>FORWARD</dispatcher>
</filter-mapping>

<filter>
 <filter-name>httpHeaderSecurity</filter-name>
 <filter-
 class>org.apache.catalina.filters.HttpHeaderSecurit
 yFilter</filter-class>
 <init-param>
  <param-name>antiClickJackingOption</param-
  name>
  <param-value>SAMEORIGIN</param-value>
 </init-param>
 <init-param>
  <param-name>antiClickJackingEnabled</param-
  name>
  <param-value>true</param-value>
 </init-param>
</filter>
```

So this turns out not to be a problem with URL Rewrite but a missing for the httpSecurityHeader filter that contained the x-frame-options. After adding the mapping "/*", every file now has the anti ClickJacking options set. Following code is showing the web.xml settings that make that happen.

```
<filter>
 <filter-name>httpHeaderSecurity</filter-name>
 <filter-
 class>org.apache.catalina.filters.HttpHeaderSecurit
 yFilter</filter-class>
 <init-param>
  <param-name>antiClickJackingOption</param-
  name>
  <param-value>SAMEORIGIN</param-value>
 </init-param>
 <init-param>
  <param-name>antiClickJackingEnabled</param-
  name>
  <param-value>true</param-value>
  </init-param>
</filter>

<filter-mapping>
 <filter-name>httpHeaderSecurity</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

A cookie's domain attribute determines which domain can access the cookie. HTTP by itself is a stateless protocol. By the use of HTTP cookies, one can differentiate between the authenticated and unauthenticated user, which further lessens the chances of hacking. Hence, the usage of insecure cookie lets the unauthenticated person access the sensitive information from the web application. The Remediation of this problem is that: By default, cookies are scoped to the issuing domain and on IE/Edge to subdomains. If one removes the explicit domain attribute from one's Set-cookie directive, then the cookie will have this default scope, which is safe and appropriate in most situations (Cookie Scoped to Parent Domain, 2019).

After reducing the issues, authors scanned the source code, again. The results of the FindBugs show the violation of 25 rules that are related to the language rule violations rather than the security violations. The result of the EasyPMD shows the 3 rule violations. The result of the SonarLint shows the run time errors that can be resolved. After implemented suggestive measurement, the result of the Arachni didn't analyze because this tool takes the open source code through GitHub. The result of the JArchitect shows the 17 rule violations related to security. The rule violations may be reduced through fixing the issues.

Vulnerabilities were present in code that we have taken from the hospital management system. We executed the tools and followed the prescriptive steps of a framework for this code of Java. The codes were written in Java; hence, Java-based tools were used for static analysis (Identifying the Exact Fixing Actions of Static Rule Violation, 2015). A before and after version of the code is available to us and new codes static analysis gives us better results. Some rule violations in the code do require to remove the software artifact on which they occur. For example, in context of a rule that detects that a method has the same implementation in a

super-class and a sub-class, we preferred ignoring them here to provide a clearer understanding of the properties of our framework.

Discussion

The increasing number of incidents on web security breach has imposed the need to look upon a direction to optimize the source code analysis to produce secure codes (Meghanathan, 2013). This practical approach is currently adopted by most of the security practitioners. In essence, the integration of security strategies as a security framework while writing the source code would allow any security anomalies to be detected and fixed well before the software application is released (El-Hadary and El-Kassas, 2014). The framework will also allow the code to be audited for conformance which, as a result, will not only provide greater security but will also save time, costs and resources which might be incurred on redevelopment or patching of the software application once it is released.

In this study, the authors have discussed the reasons for the vulnerabilities that appeared in the code and how they could be exploited if left unattended and, thus, confront with the consequences of an attack. Further, authors have provided detailed solutions to efficiently and effectively remove each of the vulnerabilities and presented the appropriate code snippets and the results of source code analysis when the vulnerabilities are fixed one after the other. FindBugs tool found 99 vulnerabilities; Arachni web application scanner found 121 vulnerabilities, SonarLint as Eclipse plugin found 100 bugs, EasyPMD as NetBeans plugin found 92 vulnerabilities. After implementation of the framework, the codes of the project were corrected following the suggestive measures and again the static analysis tools were used to get results of static source code analysis. On implementing the FindBugs tool, the vulnerability was reduced to 25. Similarly, the EasyPMD found no medium level of vulnerability. SonarLint reduces errors from 100 to 21. Hence, on successfully implementing the proposed framework, 80% of the vulnerabilities in the code are mitigated. The remaining 20% of the vulnerabilities were only compiled time errors or the vulnerabilities with low severity, whose mitigation may lead to change in the design of software.

The framework proposed and implemented in this paper proves to be relentlessly practical with the following significance:

- The framework divides the severity of the problems into three levels high, medium and low
- The results achieved by the re-analysis of code after solving the vulnerabilities issues were found to be satisfactory and low in numbers
- The framework helps to evaluate the secure source code and produce guidelines according to the severity of vulnerabilities found
- It may help to discover vulnerabilities in the software at the early stage of web application development life cycle leading to a secured end product
- It may help to determine the effects of the source code analysis for web security
- It may assist to develop alternative web security design of web application under development
- With the help of the results, developers may produce refined and prioritized coding guidelines.

Limitations of the proposed framework are as follows:

- This work used the fuzzy AHP for prioritizing the security vulnerabilities. More appropriate techniques such as fuzzy-neural, classical AHP, etc., may be used in the future that can reduce the efforts
- Security issues can be classified in more level and hierarchies to attain more security in web applications
- This framework can be applied on big projects and the results should be analyzed to validate it

As part of future work, we plan to extend the framework application with different web application based languages such as Python, Ruby and Perl. Also, the code snippet provided here is written in Java and HTML but these codes are mostly applicable to other languages as well with the same logic. Hence, removing and patching vulnerabilities with the framework is a success path for developers, which may lead to more secure web applications.

Conclusion

In this work, we have proposed and implemented a framework for securing web applications. The framework is composed of three phases with the objective of securing the source code through static code analysis. There are plenty of tools and methods that have been proposed in recent years. However, a benchmarked framework, which could combine all the tools and methods with possibly all language support, is missing. Hence, this framework combines materials and methods proposed in previous works and provide a new benchmark in web application security. To validate the proposed framework, an open source code Hospital Management Web Application system was used. The results achieved by the framework implementation are highly recognizable and satisfying. In future work, more methods other than AHP can be applied to evaluate the priorities of vulnerabilities identified.

Acknowledgment

Authors are thankful to College of Computer and Information Sciences, Prince Sultan University, KSA for providing the fund to carry out the work.

Author's Contributions

Alka Agrawal: Conceived and designed experiments, reviewed drafts of the paper, approved the final draft.

Mamdouh Alenezi: Performed the computation work, approved the final draft.

Rajeev Kumar: Conceived and designed the experiments, performed the experiments, contributed reagents/materials/analysis tools, prepared figures and/or tables, performed the computation work, approved the final draft.

Raees Ahmad Khan: Analyzed the data, contributed reagents/materials/analysis tools, authored or reviewed drafts of the paper, approved the final draft.

Ethics

The authors declare that they have no competing interests.

References

Ahmed, M. and A.S.S.M.B. Ullah, 2018. Health Care Security Analytics. In: Data Analytics, Mohiuddin Ahmed, Al-Sakib Khan Pathan (Eds.), CRC Press, ISBN-13: 9780429446177 pp: 427-440.

Ayeni, B.K., J.B. Sahalu, K.R. Adeyanju, 2018. Detecting cross-site scripting in web applications using fuzzy inference system. J. Comput. Netw. Commun. DOI: 10.1155/2018/8159548

Arachni Insecure Cookies, 2018. Available at: https://github.com/Arachni/arachni/blob/master/components/checks/passive/grep/insecure_cookies.rb

Abomhara, M., 2015. Cyber security and the internet of things: Vulnerabilities, threats, intruders and attacks. J. Cyber Security Mobility, 4: 65-88. DOI: 10.13052/jcsm2245-1439.414

Arachni Common File, 2018. Available at: https://github.com/Arachni/arachni/blob/master/components/checks/passive/common_files.rb

Arachni, 2018. Available at: https://github.com/Arachni/arachni

Beller, M., R. Bholanath, S. McIntosh and A. Zaidman, 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering, Mar. 14-18, IEEE Xplore Press, Suita, Japan, pp: 470-481. DOI: 10.1109/SANER.2016.105

Cookie Scoped to Parent Domain, 2019. Available at: https://portswigger.net/kb/issues/00500300_cookies coped-to-parent-domain

Common Directories Detection, 2017. Available at: https://www.tenable.com/plugins/was/98072

Code Optimization Control Flow Graph, 2018. Available at: http://www.utdallas.edu/~ilyen/course/compiler/notes/optimize-cfg-s.pdf

Chess, B. and G. McGraw, 2004. Static analysis for security. IEEE Security Privacy, 2: 76-79. DOI: 10.1109/MSP.2004.111

Detection of Vulnerabilities in Programs with the Help of Code Analyzers, 2008. Available at: https://www.viva64.com/en/a/0028/

Dymova, L., P. Sevastjanov and A. Tikhonenko, 2015. An interval type-2 fuzzy extension of the tops is method using alpha cuts. Knowledge-Based Syst., 83: 116-127. DOI: 10.1016/j.knosys.2015.03.014

EasyPMD, 2015. Available at: http://plugins.netbeans.org/plugin/57270/easypmd

El-Hadary, H. and S. El-Kassas, 2014. Capturing security requirements for software systems. J. Adv. Res., 5: 463-472. DOI: 10.1016/j.jare.2014.03.001

FindBugs, 2015. Find Bugs in Java Programs. Available at: http://findbugs.sourceforge.net/

Gürses, S.F. and T. Santen, 2006. Contextualizing security goals: A method for multilateral security requirements elicitation. Sicherheit, 6: 42-53.

Hospital Management System in Java Using NetBeans with Source Code, 2018. Available at: https://codeprojects.org/hospital-management-system-in-java-using-netbeans-with-source-code/

Heckman, S. and L. Williams, 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. Inform. Software Technol., 53: 363-387. DOI: 10.1016/j.infsof.2010.12.007

Huang, Y.W., F. Yu, C. Hang, C.H. Tsai and D.T. Lee *et al*., 2004. Securing web application code by static analysis and runtime protection. Proceedings of the 13th International Conference on World Wide Web, May 17-20, ACM, New York, pp: 40-52. DOI: 10.1145/988672.988679

Hussain, M., A.A. Zaidan, B.B. Zidan, S. Iqbal and M.M. Ahmed *et al*., 2018. Conceptual framework for the security of mobile health applications on android platform. Telemat. Inform., 35: 1335-1354. DOI: 10.1016/j.tele.2018.03.005

Internet Security Threat Report by Semantics, 2016. Available at: https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf

Identifying the Exact Fixing Actions of Static Rule Violation, 2015. Available at: https://hal.inria.fr/hal01185795/document

JArchitect, 2018. Available at: https://www.jarchitect.com/

Kulenovic, M. and D. Donko, 2014. A survey of static code analysis methods for security vulnerabilities detection. Proceedings of the 37th International Convention on Information and Communication Technology, Electronics and Microelectronics, May 26-30, IEEE Xplore Press, Opatija, Croatia, pp: 1381-1386. DOI: 10.1109/MIPRO.2014.6859783

Larrucea, X., I. Santamaria and R. Colomo-Palacios, 2019. Assessing source code vulnerabilities in a cloud-based system for health systems: OpenNCP. IET Software, 13: 195-202. DOI: 10.1049/iet-sen.2018.5294

Lokhande, P.S. and B.B. Meshram, 2016. Analytic Hierarchy Process (AHP) to find most probable web attack on an e-commerce site. Proceedings of the 2nd International Conference on Information and Communication Technology for Competitive Strategies, Mar. 04-05, ACM, Udaipur, India, pp: 62-62. DOI: 10.1145/2905055.2905120

Meneely, A., H. Srinivasan, A. Musa, A.R. Tejeda and M. Mokary *et al*., 2013. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Oct. 10-11, IEEE Xplore Press, Baltimore, MD, USA, pp: 65-74. DOI: 10.1109/ESEM.2013.19

Missing 'Strict-Transport-Security' Header, 2017. Available at: https://www.tenable.com/plugins/was/98056

Mardani, A., A. Jusoh, K. Nor, Z. Khalifah and N. Zakwan *et al*., 2015. Multiple criteria decision making techniques and their applications–a review of the literature from 2000 to 2014. Economic Res., 28: 516-571. DOI: 10.1080/1331677X.2015.1075139

Mu, E. and M. Pereyra-Rojas, 2017. Understanding the Analytic Hierarchy Process. In: Practical Decision Making, Springer, ISBN-13: 978-3-319-33860-6, pp: 7-22.

Meghanathan, N., 2013. Source code analysis to remove security vulnerabilities in java socket programs: A case study. Int. J. Netw. Security Applic., 5: 1-16. DOI: 10.5121/ijnsa.2013.5101

Nausheen, F. and S.H. Begum, 2018. Healthcare IoT: Benefits, vulnerabilities and solutions. Proceedings of the 2nd International Conference on Inventive Systems and Control, Jan. 19-20, IEEE Xplore Press, Coimbatore, India, pp: 517-522. DOI: 10.1109/ICISC.2018.8399126

Nunes, P., I. Medeiros, J.C. Fonseca, N. Neves and M. Correia *et al*., 2018. Benchmarking static analysis tools for web security. IEEE Trans. Reliability, 67: 1159-1175. DOI: 10.1109/TR.2018.2839339

Perl, H., S. Dechand, M. Smith, D. Arp and F. Yamaguchi *et al*., 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Oct. 12-16, ACM, Denver, Colorado, USA, pp: 426-437. DOI: 10.1145/2810103.2813604

Paladion, Source Code Analysis Suite, 2018. Available at: https://www.paladion.net/blogs/source-codeanalysis-suite

Pistoia, M., S. Chandra, S.J. Fink and E. Yahav, 2007. A survey of static analysis methods for identifying security vulnerabilities in software systems. IBM Syst. J., 46: 265-288. DOI: 10.1147/sj.462.0265

Strict-Transport-Security' Header, 2018. Available at: http://www.valencynetworks.com/kb/clickjacking-xframe-options-header-missing.html

Smith, J., B. Johnson, E. Murphy-Hill, B.T. Chu and H. Richter, 2018. How developers diagnose potential security vulnerabilities with a static analysis tool. IEEE Trans. Software Eng., 45: 877-897. DOI: 10.1109/TSE.2018.2810116

Saaty, T.L., 1985. Decision making for leaders. IEEE Trans. Syst. Man Cybernet., 3: 450-452. DOI: 10.1109/TSMC.1985.6313384

Standard, D., 1997. Requirements for safety related software in defence equipment part 2: Guidance. http://www.software-supportability.org/Docs/00-55_Part_2.pdf

Sonarlint, 2017. Fix Issues before They Exist.

The Autocomplete Attribute and Web Documents using XHTML, 2011. Available at: https://wiki.mozilla.org/The_autocomplete_attribute_and_web_documents_using_XHTML

Unvalidated Redirection, 2018. Available at: https://wordpress.org/support/topic/unvalidated-redirection/

VanDen H., B. Jeroen, P.W. Martijn and M. Warnier, 2018. Privacy and information technology.

Visual Code Grepper - Code Security Scanning Tool, 2016. Available at: https://github.com/nccgroup/VCG

Verma, A.K. and A.K. Sharma, 2019. An Assessment of Vulnerable Detection Source Code Tools. In: Software Engineering, Hoda, M., N. Chauhan, S. Quadri and P. Srivastava (Eds.), Springer, Singapore, ISBN-13: 978-981-10-8847-6, pp: 403-412.

Web Application Vulnerabilities: Statistics for 2017, 2018. Available at: https://www.ptsecurity.com/wwen/analytics/web-application-vulnerabilities-2018/

Yamaguchi, F., N. Golde, D. Arp and K. Rieck, 2014. Modeling and discovering vulnerabilities with code property graphs. Proceedings of the IEEE Symposium on Security and Privacy, May 18-21, IEEE Xplore Press, San Jose, CA, USA, pp: 590-604. DOI: 10.1109/SP.2014.44

Zampetti, F., S. Scalabrino, R. Oliveto, G. Canfora and M. Di Penta, 2017. How open source projects use static code analysis tools in continuous integration pipelines. Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories, May 20-21, IEEE Xplore Press, Buenos Aires, Argentina, pp: 334-344. DOI: 10.1109/MSR.2017.2.