International Journal of Advances in Computer Networks and Its Security
Volume 6 : Issue 2 [ISSN 2250-3757]
Publication Date : 31 August, 2016

Dynamic Analysis for Security Testing of WEB Based Applications Using Agent Technology

[Muhammad Imran, Fathy Eassa, and Kamal Jambi]

***Abstract*—This is the second part of two papers covering the topic of security testing of WEB based applications using agent technology that covers dynamic analysis. This research covers an integrated dynamic analysis technique and tool for detecting and preventing such security vulnerabilities in web applications. It is based on agent technology and written in Java. The dynamic analysis starts for tracking the propagation of user input in the program which helps to detect the vulnerabilities in the source code. This technique is extendable to the vulnerabilities in the similar class and source codes written in other object oriented languages. At the end, this paper presents a Java Web Application Security Tester (JWAST) which is an implementation of the proposed technique. Also a comparison of JWAST with other tools is presented.**

***Keywords*—Dynamic Analysis, Security Testing, WEB Based Applications, Agent Technology, Software Engineering**

I. Introduction

Static analysis can be defined as “the analysis of a computer software which is performed without actually executing the software under testing” [1]. The program’s text is statically examined in this analysis and a possibility of applying the static analysis on the compiled form of the program also exists but decoding can be a problem in this case. Manual auditing and Code Review might also fall under the category of static analysis but this remains ineffective until and unless this activity is automated making it faster and reliable.

Runtime monitoring falls under the category of dynamic analysis and this refers to monitoring the program under test during its execution. Further, different techniques exist for achieving the runtime monitoring. However, the techniques of runtime monitoring are more feasible for preventing the runtime attacks rather than their detection. So, it cannot be used for detecting the location of vulnerabilities but it can help in preventing the vulnerability to be exploited during the program execution.

Similarly, another set of techniques that also falls under the category of dynamic analysis is known as penetration testing [2]. In penetration testing the system is considered a black box. A set of input variables, whose values are set as some malicious inputs, is composed manually or automatically and is given to the program under test. And finally the behavior of the program as a result of that input is evaluated. However, this technique depends on the set of input values that are given and setting these input requires a significant security knowledge.

Muhammad Imran, Fathy Eassa, and Kamal Jambi

Computer Science Department, King Abdulaziz University
Jeddah, Saudi Arabia.

This paper starts with an introduction of dynamic analysis followed by a detailed section of the proposed agent based security testing tool called Java Web Application Security Tester (JWAST). At the end there is a coverage comparison of JWAST with other tools.

II. Dynamic Analysis

In the literature, several dynamic data flow analyses have been proposed, researched and used in the software security solutions and tools. In a dynamic taint analysis, the program is executed and those computations and method calls are searched which are affected the un-validated user input. Different works targeting the security vulnerabilities problem follows this approach. One of these works include a dynamic taint analysis approach by Haldar [3]. This approach targets the Java web applications and looks for XSS, Cookies Poisoning and Command injection attacks at the runtime.

One of the approaches [4], prevents the SQL injection attacks during the runtime by building a parse tree of the SQL statements before including the user input and after including the user input in the SQL statement and comparing both of them. By comparing the before and after user input parse trees, it is decided whether the structure of the statement is same and attack is attempted or not. One other similar approach based on the parse trees building at the runtime is used in the tool called CANDID [5]. It uses its runtime analysis for preventing the SQL injection attacks. The attacks are prevented by recording the sequence of SQL commands and replacing the inputs in these commands with 1s and then building the parse tree. If the parse tree differs from the original parse tree, the query is preventing from execution.

An approach from Boyd [6] uses a functionality provided by Java for preventing the SQL injection attacks. They use the PreparedStatement API available in Java and forces the SQL queries to be containing only string or numeric literals. Also the SQL keywords are randomized, so that they could not be guessed by the users however, this is the limitation in this approach also, as it would be compromised it user guesses the randomization key successfully.

Another dynamic taint analysis approach by Chang et al. [7] is targets the C programs and looks for the command injection attacks and format string attacks. In this approach, a data flow analysis is integrated at the compile time using a small library that tracks the taints throughout the program during execution for detecting the vulnerabilities that are caused due to inputting the untrusted data.

The dynamic taint analysis is a common approach to follow for preventing the attacks at the runtime time. Several works which include [8] and [9] used this form of analysis. The dynamic taint analysis is actually influenced by the Perl’s taint mode. Also, one other way to dynamically prevent the attacks from happening is the use of wrappers [10]. The wrapper to the program will filter out the malicious input values which will eventually prevent malicious input to reach the actual program and the security vulnerabilities would not be exploited.

The dynamic taint analysis is not only used for Java, but Salvatore [11] also used it for tracking the taints information at the character level in the PHP programs. In this proposed technique, the SQL query is tokenized and checked for the existence of any tainted values in it. Similarly, Wasp [12] uses tainting technique for Java by providing the bytecode instrumentor and tainting the strings.

A. *Integration of Static and Dynamic Analysis*

There are pros and cons for both the static and dynamic analysis techniques where the static analysis is able to do high code coverage with low accuracy and dynamic analysis is opposite to that. To neutralize their cons and maximize their pros, the integration of these two techniques has been the subject of this project and similarly several other researches in the past have used this concept in their solutions. A very simple example for the integrated technique is the technique used to prevent the XSS attacks on the client side, by combining the static and dynamic analysis in a web browser [13].

The work by Lucca et al. [14] is based on identifying Cross-Site Scripting vulnerabilities in web applications. It presents and approach which is combination of Static and Dynamic analysis where static analysis is supposed to detect potential vulnerabilities and then the dynamic analysis will help in detecting the actual vulnerabilities. To prevent the XSS attack one of the recommended solution is to disable the scripting languages in the bowser however this problem should be addressed by the developers instead on end users. Another option suggests to use the input validation functions after each input but this will result in an overhead as all the inputs might not affect the output data which will not cause XSS. This work proposed an approach to analyze only the input data which affects the output data for which it exploits both static and dynamic analysis. They used some predicates to define some rules by applying them to the Control Flow Graph (CFG) of the server page for assessing its vulnerability. By using the predicates in some conditions the vulnerabilities are characterized as Potentially Vulnerable (PV), Vulnerable with respect to v (V) and not vulnerable (NV). For the dynamic analysis, output of the static analysis is exploited by submitting only those pages which were found vulnerable in the static analysis. For the dynamic analysis the author defines a set of XSS attack strings and for each string it executes each vulnerable server page by giving the attack string as input to each vulnerable field of that page, after which the attack consequences are checked. To test the effects of the attack in the dynamic analysis it might be difficult when the output/malicious data is not provided to the user but stored in the database. Thus to observe the effects of XSS WATT (Web Application Testing Tool) has been used which takes the input from the XSS test case generator module and the results of test case execution are checked to assess the success of the attack.

An integrated technique is used in a tool Saner [15]. It detects the sanitization routines in a program with a static analyzer based on the already existing tool called Pixy [49]. After the static analysis is done, the dynamic analysis is used making the tool more sound and complete by checking if the detected sanitization is correct and complete.

Amnesia [16] integrates the static and dynamic analysis and used to prevent the SQL injection attacks at the runtime. The static analysis is performed in this tool by building a model of valid SQL queries and then in the dynamic analysis the queries generated at the run time are checked against the statically built model that whether these runtime queries comply with the statically built model.

One of the integrated techniques is proposed for detecting the security vulnerabilities in the PHP based web applications. This technique is used by a tool known as WebSSARI [17]. The static analysis of WebSSARI constructs the Abstract Syntax Tree, Control Flow Graph and uses then to track the state of variable in the program with the help of a symbol table. The path between the taint values and dangerous functions is identified and the next part is done for the runtime prevention and detection of attacks. Specific instrumentation code is inserted based on the static analysis results and this code performs checks and prevents the security attacks during the application runtime

III. JAVA Web Application Security Tester

This section covers the architecture of our agent based security testing tool JWAST.

A. *The Testing Methodology*

This section presents the technique built for the purpose of security testing of web based applications. As shown in [1] various security testing techniques can be used to test the software programs for known or unknown security vulnerabilities.

JWAST is a static and dynamic testing tool which is based on integrated static and dynamic analysis technique. The basic idea behind this technique is the use of static analysis technique and integrating it with a dynamic analysis technique to increase the detection capability of our tool and also enable our tool to prevent the attacks from happening at the run time [1].

B. *Dynamic Analysis Technique*

The tool starts by the task of code analysis where static analysis is performed first. The static analysis operates on Java source code files where it analyzes every file to determine specified vulnerabilities. The vulnerabilities are specified by the security rules which behave as the security knowledge for static analysis technique [1]. Once the static analysis is completed, the next step is to perform the dynamic analysis on the web application. The dynamic analysis carries out the testing process by the use of instrumentation technique. The instrumentation approach is based on the idea that, the attacks occurring due to the input validation vulnerabilities can be handled by adding the validation to the source code by determining of instrumentation technique of the original source code with the pre-defined instrumentation templates. Therefore, the instrumentation code would perform the validation on the

International Journal of Advances in Computer Networks and Its Security
Volume 6 : Issue 2 [ISSN 2250-3757]
Publication Date : 31 August, 2016

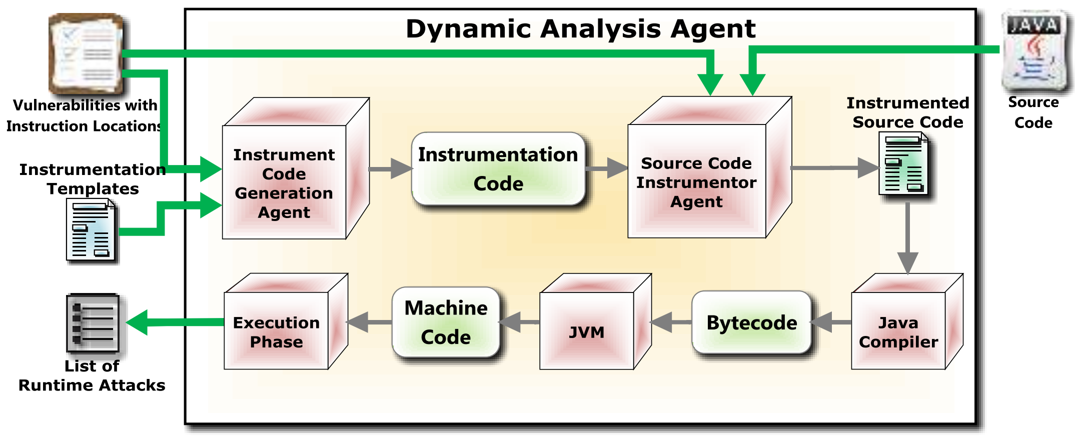

Figure 1: Dynamic Analysis Agent Architecture

input given at the runtime, as a result of which the attacks would be stopped from being carried out and also the attempt for an attack can be reported during the web application's runtime.

To do this, an automated dynamic analyzer agent generates the instrumentation code based on the instrumentation templates that contains the specified templates for each target vulnerability type. Later on, the dynamic analyzer agent also inserts the generated instrumentation code into the original web application code automatically. For inserting the instrumentation code, the locations are extracted from the results produced by the static analyzer agent. As, the instrumented source code, which is actually combination of the original source code and the instrumentation code, is executed the runtime attacks are prevented as well as reported to the user.

C. *The Architecture*

As in the previous section, we have presented the high level architecture of our tool. This section presents the low level architecture of our tool in details where each agent's architecture is presented and described in detail.

1) Dynamic Analysis Agent Architecture

The dynamic analysis agent is responsible for carrying out the dynamic part of our testing tool. Similar to other analysis agents involved in the testing process, the dynamic analysis agent also takes several inputs that are consumed by different agents acting as sub agents for the dynamic analysis agent.

The architecture diagram of the dynamic analysis agent is shown in Fig. 1. The operations performed by the dynamic analysis agent are described in the points below:

1. The list of vulnerabilities is given as an input, along with the predefined instrumentation templates, to the instrument code generation agent.

2. The instrument code generation agent generates appropriate instrumentation code based on the vulnerabilities information provided by the vulnerabilities list. This information mainly includes the types of potential vulnerabilities, the location of vulnerabilities in the source code and the vulnerable method along with the vulnerable parameter of that method.

3. The instrumentation code is passed on to the source code instrumentor agent. This agent also takes the source code of the web application under test and then instruments that code by adding the instrumentation code at appropriate locations.

4. The instrumented source code generated by the source code instrumentor agent is given to the Java compiler which compiles it and produces the bytecode.

5. Bytecode is taken as input by the Java Virtual Machine (JVM), and machine code is produced, which further goes through the execution phase.

When the web application which has already been instrumented, runs during the execution, the runtime attacks are detected, prevented and a list of these attacks is generated as a final output of the tool.

D. *Implementation and Testing*

This section covers the implementation details followed by the testing that we have performed on our security testing tool JWAST.

2) General View

The implementation of the tool is done in the Java language in the form of independent agent based subsystems. All agents of JWAST are written in the Java programming language and for developing, managing and running the agents, JADE framework version 4.3.2 is used as a middleware [1]. The implementation of the tool is done in two phases based on the idea used in the proposed integrated static and dynamic analysis technique. In the first phase, the static analysis is implemented which further consists of subsystems that interact with each other and take the output of one or more subsystems as their input. In the second phase of implementation the dynamic analysis has been implemented based on the results and their format produced during the static analysis. While implementing the static and dynamic analysis modules, a specific mobile agent framework is used.

3) Implementing Dynamic Analysis

For implementing any static code analysis technique, the preprocessing (performing lexical analysis and parsing) is

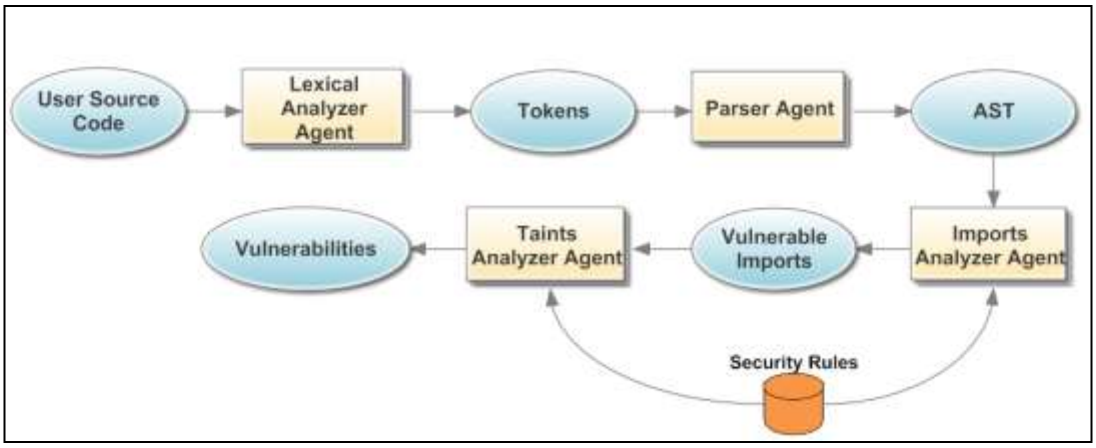

Figure 2 : JWAST test case design diagram

TABLE I. COVERAGE COMPARISON OF JWAST WITH OTHER TOOLS

Vulnerabilities/Tools	JWAST	PMD	Find Bugs	RIPS	SAFELI
SQL injection	✓	✓	✓	✓	✓
Cross-Site Scripting	✓	✓	✓	✓	×
Http Response Splitting	✓	×	×	✓	×
Path Traversal	✓	×	×	×	×
Command Injection	✓	×	✓	✓	×
XPath Injection	✓	×	×	×	×
LDAP Injection	✓	×	×	×	×

typically required and generally the manual construction of lexical analyzer and parsers is rare [1]. Thus a single or a set of tools, depending on the technique under development, are used for automating the lexical analyzer and parser construction.

Once the static analysis is completed and the vulnerabilities along with all the details are produced as a result, the dynamic analyzer agent takes control. The implementation of the dynamic analysis is further divided into two modules as in the JWAST architecture shown in Fig. 1. The instrument code generation agent implements the traversing and analysis logic. The traversing part implements the logic for visiting the instrumentation templates that are input to the dynamic analysis module. This traversing is performed on the basis of analysis that runs over each vulnerability to identify the vulnerability type and extracts the required parameters like vulnerability location in the source code. Once the vulnerability is identified, the specific instrumentation template is selected and instrumentation method from the source code instrumentor agent is called where it is provided with the instrumentation code and the source code where this is instrumentation code is to be inserted. For instrumenting the source code, the source code processor called Java Instrumentation Engine (JIE) version 1.01 [18] , meant for source code instrumentation, is used. “The Java Instrumentation Engine (JIE) is a generic Java source code processor which inserts instrumentation code at specified locations in a given source code. In its basic mode of operation, JIE receives a Java source file and instrumentation instructions, and emits appropriately transformed Java source code” [18]. Once the source code is instrumented, it is converted to byte code by the Java compiler and then into the machine code by the Java Virtual Machine (JVM) and in the execution phase the instrumentation code gets executed where the runtime attacks are then detected and prevented.

There exist a number of tools that are meant for security testing. Here, we present a coverage comparison of our tool with some of the other tools built for the web application security testing. Table 1 shows the difference in the vulnerability coverage that exists between our tool and other tools. As it shown in the table, our tool JWAST covers all the listed vulnerabilities. JWAST is capable of detecting and preventing the SQL injection, XSS, Http response splitting, path traversal, command injection, XPath injection and LDAP injection vulnerabilities and runtime attacks [1]. However, the other tools that we have compared with, cover only few of these vulnerabilities where path traversal, XPath injection and LDAP injection vulnerabilities and attacks can only be detected using our tool, JWAST.

TABLE II. FEATURE COMPARISON OF JWAST WITH OTHER TOOLS

Features/Tools	JWAST	PMD	Find Bugs	RIPS	Dytan
Underlying Technology	Agent Based	Conventional Desktop Based	Conventional Desktop Based	Conventional Desktop Based	Desktop Based
Techniques Used	Static + Dynamic Analysis	Static Analysis	Static Analysis	Static Analysis	Dynamic Analysis
Target Language	JAVA	JAVA	JAVA	PhP	binaries
Input Format	Source Code	Source Code	Byte Code	Source Code	x86 binaries
Extensibility	Yes	Yes	Yes	No	Yes

TABLE III. RESULTS OF TEST RUN ON JULIET TEST CASE VERSION 1.1.1 AND VERSION 1.2

Juliet Test Case	**Total Test Cases**			**Precision**	**Accuracy**	**Recall**
Version 1.1.1	6330	True Positives	1250	**0.13**	**0.6**	**0.2**
		False Positives	8421			
		True Negatives	18892			
		False Negatives	5062			
Version 1.2	9731	True Positives	2008	**0.4**	**0.8**	**0.2**
		False Positives	3180			
		True Negatives	40932			
		False Negatives	7723			

There is a comparative study that we have made for our tool with the other existing tools. In Table 2, we have shown the differences in the implemented features of our tool with other tools. This comparison shows that our tool has contributed in terms of several improvements in the features that current tools offer.

As it can be seen in the Table 2, JWAST differs in the underlying technology, where we have introduced an agent based security testing tool as compared to the other existing tools which developed as a conventional desktop based tools [1]. Also, the technique that we have introduced and used in our tool is an integrated static and dynamic analysis based technique which is in contrast to the other tools that use either only static or only dynamic analysis techniques for the testing. One other obvious advantage of our tool over the other tools is the input format used for performing the security testing, which is the source code instead of the binaries or the bytecode. This enable the users to perform the security testing even if the application is under development and the bytecode or binaries are not available yet.

4) Evaluation Results

For evaluating JWAST, we used tests suites provided by Software Assurance Metrics And Tool Evaluation (SAMATE) [19]. For the testing the functionality of JWAST, we have used White-box and Black-box testing techniques. In white box testing, internal code written in every component was tested and it was checked that the code written is efficient in utilizing the resources of the system like memory, band width or the utilization of input/output. In order to perform Black-box testing on the tool, we prepared several formal test case pairs for each type of the vulnerabilities. Each pair in the formal test cases consisted of negative and positive tests where, a negative test is to be performed on the non-vulnerable code and positive test is to be run on the vulnerable code known in advance. The test cases that we have run are designed for different vulnerabilities and are according to the Fig. 2. They provide several test suites for performing security tools evaluation and we have used the Juliet Test Suite for Java version 1.1.1 and Juliet Test Suite for Java version 1.2. Table 3 shows the summary of the results that we have obtained by running JWAST on the Juliet Test Suite version 1.1.1 and the Table 3 shows the results that are obtained by the Juliet Test Suite version 1.2.

IV. Conclusion

In this research, the topic of security testing of WEB based applications using agent technology that covers dynamic analysis is covered. The tool is implemented in Java as an agent based security tool for testing the web based applications written in Java. We have conducted several experiments to test the ability of JWAST to detect input validation vulnerabilities. We designed and run several positive and negative test cases for each type of vulnerability in the input validation vulnerabilities class. The results have shown that our tool detects and prevents the input validation vulnerabilities and is sound with respect to its rule base.

A comparison of JWAST with other existing tools revealed that JWAST performs better than the other tools. JWAST provides improved coverage in terms of support for number and types of security vulnerabilities as compared to the other tools. Also, the features that are provided by JWAST are better than the other tools. Namely, the underlying technology, the integrated testing technique and the input format as a source code are the features where JWAST takes an edge over other tools.

References

[1] M. Imran, F. Eassa, and K. Jambi, "Using Agent Technology for Security Testing of WEB Based Applications", (SEDE–2015). P. 3-10, San Diego, California, USA, 2015.

[2] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," IEEE Secur. Priv., vol. 3, no. 1, pp. 84–87, 2005.

[3] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for Java," in Computer Security Applications Conference, 21st Annual, 2005, p. 9–12.

[4] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," in International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC, 2005.

[5] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: preventing sql injection attacks using dynamic candidate evaluations," in Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp. 12–24.

[6] S. Boyd and A. D. Keromytis, "SQLrand: preventing SQL injection attacks," in Applied Cryptog- raphy and Network Security Conference, 2004.

[7] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in Proceedings of the 15th ACM conference on Computer and communications security, 2008, pp. 39–50.

[8] W. Xu, S. Bhatkar, and R. Sekar, "Practical Dynamic Taint Analysis for Countering Input Validation Attacks on Web Applications" Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, 2005, pp. 1–15.

[9] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software", https://scholar.google.com/scholar?cluster=9428084648194964145&hl=en&as_sdt=0,5, 2005.

[10] D. Scott and R. Sharp, "Abstracting application-level web security," in Proceedings of the 11th international conference on World Wide Web, 2002, pp. 396–407.

[11] A. N. Salvatore, G. Doug, and G. David, "Automatically Hardening Web Applications Using Precise Tainting", Springer US, 2005.

[12] William G. J. Halfond Alessandro Orso and P. Manolios, "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks," in ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2006), 2006.

[13] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis", Proc. of the Network and Distributed System Security Symposium (NDSS'07), 2007.

[14] G. A. Di Lucca, A. R. Fasolino, M.Mastroianni, and P.Tramontana, "Identifying Cross Site Scripting Vulnerabilities in Web Applications", Springer US, 2005.

[15] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, and C. K. G. Vigna, "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," IEEE Secur. Priv., 2008.

[16] Halfond, W. GJ, and A. Orso, "AMNESIA : Analysis and Monitoring for NEutralizing SQL-Injection Attacks," in 20th IEEE/ACM international Conference on Automated software engineering, 2005.

[17] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," Proc. 13th Conf. World Wide Web - WWW '04, p. 40, 2004.

[18] E. Tromer, "Java Instrumentation Engine." [Online]. Available: http://cs.tau.ac.il/~tromer/jie/. [Accessed: 05-Feb-2014]

[19] NIST, "SAMATE." [Online]. Available: http://samate.nist.gov/Main_Page.html. [Accessed: 07-Feb-2014].

About Author (s):

Muhammad Imran received Master degree from king Abdulaziz University (2014). His area of interest covers: software engineering, and agent technology.

Fathy Eassa received Ph.D. from University of Colorado, Boulder, U.S.A (1989). His area of interest covers: software engineering, agent technology, and Big data.

Kamal Jambi received Ph.D. from Illinios Insitute of Technology, Chicago, U.S.A (1991). His area of interest covers: software engineering, agent technology, speech recognition, image processing and OCR.