IJCTA, 9(30), 2016, pp. 313-320
© International Science Press

Multilevel Metamodel for Heuristic Search of Vulnerabilities in the Software Source Code

Alexey Sergeevich Markov* Andrei Anatolievich Fadin and Valentin Leonidovich Tsirlov*****

Abstract : This paper is devoted to the structural static analysis of the source code security and the solution to the problem associated with completeness of checks. To ensure the completeness of checks to detect vulnerabilities in the source code, the use of heuristic (signature) method for analysis of software security, which takes into account a full range of classes of software security weaknesses, is justified. A semantic metamodel for the description of heuristic algorithms for detecting software security vulnerabilities and weaknesses at different levels of the source code presentation has been developed. It is noted that the most prospective models for software presentation from the point of view of security are an abstract syntax tree and an abstract semantic graph. It is shown that the necessary level of a response rate, formal simplicity and visualization of the heuristic analysis can be achieved by the use of a production system. Examples of particular semantic models of heuristics for identifying actual classes of software security weaknesses are presented. Advantages and limitations of the solutions offered are noted. Data on implementation and approval of proposed solutions in practice are provided. It is pointed out that, in certification testing of information protection systems, 88% of critical vulnerabilities were identified by the heuristic analysis. It was concluded that the heuristic analysis may form a basis for different methods of the source code security audit.

Keywords : Information security, software security, testing, static analysis, production models, heuristic analysis, vulnerabilities, weaknesses, defects, undocumented features.

1. INTRODUCTION

When testing software systems according to the security requirements, testing laboratories face the fundamental challenge - confirmation of performed checks completeness. The solution to this problem is not trivial due to the use of different testing methods oriented to certain classes of vulnerabilities. For example, the use of traditional methods for fuzz-testing is not very effective in detecting vulnerabilities related to rare combinations of input data, for example, software bugs [1]. Applied approaches to verifying software based on formal (logic-algebraic and executable) models [1-10] are limited to a narrow range of vulnerabilities, as non-functional errors (incorrect coding).

This paper offers an applied approach to the heuristic analysis of the software security which ensures completeness and consistency of checks due to the support for known registers of the code security weaknesses [11, 12].

The relevance of the theme should be noted, as requirements to mandatory analysis of the source code security are stated in the popular Common Criteria method addressing the highest evaluation levels of credibility and PCI DSS, PA-DSS, NISTIR 4909. The above is deemed to be confirmed by a new international project in the area of the static analysis - SATEC [13].

* Bauman Moscow State Technical University, Russia, 105005, Moscow, 2-nd Baumanskaya, 5, 1
** NPO Echelon, Russia, 107023, Moscow, Elektrozavodskaya ul., 24
*** Bauman Moscow State Technical University, Russia, 105005, Moscow, 2-nd Baumanskaya, 5, 1

Today, due to critical complexity of software, it is not possible to check the code security without an expert. The key objective of automated vulnerabilities search is the provision to the expert of the code tags with a high degree of probability that it contains a vulnerability of a certain class. In this case, a static analyzer does not manipulate vulnerabilities but weaknesses in the code security themselves [14], which are understood as defects in the software design that are likely to affect the degree of information security [15-17]. An exploited defect in security is considered to be a vulnerability which use at the information facility poses information security of a certain resource at risk (Figure 1).

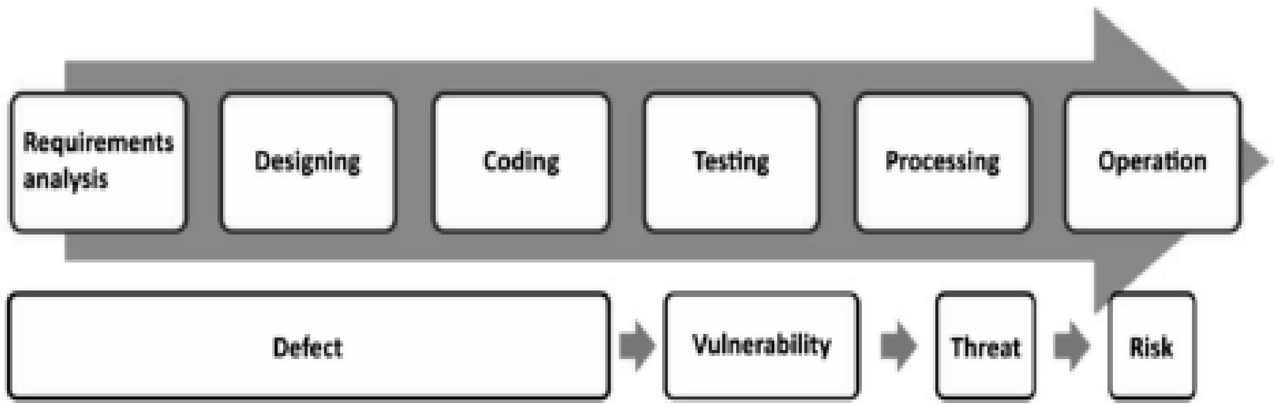

Fig. 1. Information security factors over the software life cycle.

The heuristic analysis means a search of the software weaknesses in the source code by comparing code fragments with samples from the database of security weaknesses templates (heuristic rules).

Apart from the software source codes, the targets of analysis can be object and executable software codes, project, compilation, setting and layout information [18-20]. The analysis can be conducted with different models for code presentation (depending on the level of unification), such as lexical code, syntax tree, abstract syntax tree (Kantorovich's tree), control flow graph, data flow graph, single static assignment model, abstract semantic graph (Boulanger, 2012; Chess and West, 2007).

To ensure completeness of checks, it is necessary to map the templates database to up-to-date registers of software weaknesses, such as: CWE, HP Fortify, DoD SFP, WASC-T, etc.

Table 1 compares static analysis methods: applied verification method (properties check) and heuristic analysis method (search by template) at the most popular level of the code presentation.

Table 1. Compared methods for vulnerabilities search.

Code presentation	*Properties check*	*Weaknesses search by template*
Source codes	Unavailable	Searching signatures by regular expressions
Abstract syntax tree	Lexical and syntax analysis	Lexical and syntax analysis, searching signatures by the tree
Abstract semantic data flow graph	Abstract interpretation (interval analysis, index analysis, data dependence analysis)	Data flow analysis, searching in-procedure signatures of instructions sequence with regard to values transmitted

Improvement in the performance of the code static analysis is conditioned by the decrease in the number of false positives with a complete set of identified classes of potentially unsafe constructions. The apparatus that describes templates (signatures, heuristics) of the code weaknesses must deliver ultimate flexibility in detecting weaknesses with regard to diverse syntax of the certain programming language, which is deemed to be the primary objective of the research.

2. METHODOLOGY

2.1. Developing a Metamodel for Defining Code Weaknesses Security

To simplify an overview of the metamodel, it is recommended to use a knowledge production model. This is due to that today there are not more than 100 known classes of critical weaknesses associated with a certain programming language (Barabanov, Markov, Fadin and Tsirlov, 2015). The following system of productions is proposed:

$$\begin{cases} \mathrm{R}_1 = < s_j \in \mathrm{S}\,; \mathrm{CP}_1\ (s_j\ )\,; \mathrm{DA}_1(s_j\ ) \rightarrow \mathrm{DB}_1(s_j\ );\ \mathrm{P}_1\ (\mathrm{CL}_1, \mathrm{PR}_1\ ) > \\ \vdots \\ \mathrm{R}_n = < s_j \in \mathrm{S}\,; \mathrm{CP}_n\ (s_j\ )\,; \mathrm{DA}_n(s_j\ ) \rightarrow \mathrm{DB}_n(s_j\ );\ \mathrm{P}_n\ (\mathrm{CL}_n, \mathrm{PR}_n\ ) > \end{cases}, \tag{1}$$

where n is a number of production rules for code analysis in the model;

R_i is i-th number of the model product, $i = \overline{1,n};$

$s_j \in \mathrm{S}$ is a description of a class of cases (in this area of interest we are referring to the check of the analyzer's ability to process input data);

S is sets of all lexical tokens of the software project source codes, which can be processed by this analyzer (the restriction is set by the programming language, its grammar, type of supported signatures for analysis);

$j = \overline{1,k}$ is a lexical token index in an array of lexical tokens $\bar{s}$, which is currently processed by the static analyzer, $k = |\bar{s}|$; $\bar{s}$ is an array of lexical tokens, nodes of the code abstract syntax tree resulting from the preprocessor operation, lexical and syntax analysis;

$\mathrm{CP}(s_j)$ is a condition for activating products, the function which returns 1 with available standard construction of the software project code set in the argument, where a software weakness (of a predefined type) is most likely to occur or where software features important for the analysis are available (a certain type of the functional object, constant or statistically predictable value, etc.);

$\mathrm{DA}(s_j) \rightarrow \mathrm{DB}(s_j)$ is an expression, a so called core of the production with a production output about a code weakness or additional code properties based on the existing standard syntactic code constructions which determine whether any weakness may occur or whether there are any code properties that indirectly bring about a weakness . The output may be either intermediate or final;

P(CL,PR) is a function called after a logical conclusion on the existence of a potential code weakness is reached. This function has the following parameters:

CL is a category (type) of the weakness. This parameter is set by a code audit expert when creating a production model. The category is a number assigned to the weakness type in the international vulnerability classifications;

PR is a criticality level of a given weakness. This parameter is set by an expert.

The parameters of the above functions are different types of functional and information objects received after structural presentation of the code.

The method for operation of the proposed production models system includes the following steps:

1. The s_j token is read out (originally j has an index of the first element of lexical tokens tuple);
2. All model productions are checked sequentially (1,…, n);
3. If neither is activated in cycle 2, than increase j by 1, and go to item 2;
4. If there are no more tokens in the input flow than complete the work, otherwise go to item 1 (Figure 2).

This system of productions is a multilevel metamodel for describing heuristics of searching certain vulnerabilities as demonstrated below.

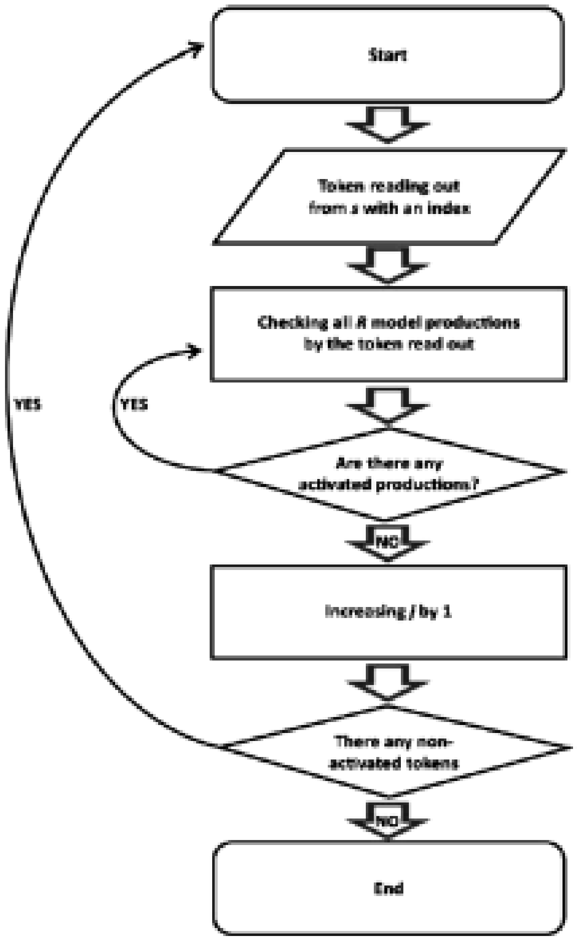
Start
Token reading out from a with an index
Checking all R model productions by the token read out
YES
YES
Are there any activated productions?
NO
Increasing j by 1
There any non-activated tokens
NO
End

Fig. 2. Flow chart of the production models system performance.

2.2. Methods for Heuristics Formal Characterization for Certain Classes of the Code Weaknesses

In the description of certain heuristics we have to rely upon certain classes of weaknesses, a certain programming language and level of the software code presentation [14,21]. As a detailed description of all sorts of heuristics is associated with a large volume and complex perception [22], we will have to restrict ourselves to several examples.

Let us introduce the following definitions and symbols.

The functional object should be understood as a software component which processes a complete fragment of the software operation algorithm. Functions, procedures, classes and objects may serve as functional objects.

The set of API FO should be understood as a set of functional objects of standard libraries for programming environment, operating system, and external components (libraries, frameworks), which ensure the implementation of a given type of functionality [23].

We will define a tuple of allowed types of objects in the software (procedural and object in nature) as follows:

$$M = <M_{const}, M_{func}, M_{info}, M_{obj}, I_{det}>, \quad (2)$$

where M_{const} is a set of admissible constant values in the software (literals, constants, macros, etc.);

M_{func} is a set of functional objects (FO) defined in the software system;

M_{info} is a set of information objects (IO) defined in the software system;

M_{obj} is a set of objects (classes) defined in the software;

I_{det} is a set of IO with a statistically predictable value (system time, process identifiers, and any constants).

The tuple of allowed types of external functional objects (API) in the software is defined as follows:

$$F = < F_{prng}, F_{sql} >, \quad (3)$$

where F_{prng} is a set of API functions which initialize pseudorandom-number generators;

F_{sql} is a set of API functions which process SQL queries.

The tuple of admissible types of operands in the software is defined as follows:

$$o = < o_{coint}, o_{call}, o_{ret}, o_{asg} (V, R) >, \quad (4)$$

where o_{coint} is a set of operators and functions responsible for integer object conversion;

o_{call} is a set of operators and functions responsible for the control transfer to the functional object;

o_{ret} is a set of operators and functions responsible for the control transfer from the functional object;

o_{asg} (V,R) is a set of operators and functions responsible for the assignment of values typical for a set of information objects V to the information object R.

The tuple of admissible functions over the code structural presentation is defined as follows:

$$D = < f_{inside} (b), f_{call} (b), t_{obj} (b) >, \quad (5)$$

where $f_{inside} (b)$ is a function that returns a set of functional and information objects inside b;

$f_{call} (b)$ is a function that returns a tuple of arguments used to launch a functional object or initialize an information object depending on the type of b;

$t_{obj} (b)$ is a function that returns b, if it was set earlier in the software source code; if no type is determined at this stage of analysis (for example, a programming language with dynamic typing is used), it returns nil.

L is determined as a set of lexical tokens with potentially unsafe constructions (this set is completed while the production model is in operation) and AP(b,e) is a function that adds b to the set e.

We determine rules for production output as related to weaknesses of different types.

3. MODEL IMPLEMENTATION RESULTS

3.1. Examples of Heuristic Analysis Based on the Production Model

The above metamodel can be implemented as an arbitrary number of production models, which detect the code weakness of various types. ?n in-procedural level of the code presentation is used (abstract syntax tree) to identify a potentially unsafe fragment. Examples of weaknesses are given in Table 2.

Table 2. Examples of the code security weaknesses.

Upper level CWE	***Lower level CWE***	***Presence bit***	***Approved programming languages and technologies***
Use of Insufficiently Random Values, CWE-330	Predictable Seed in PRNG, CWE-337	Use as a seed for initialization of constants or predictable values	No dependence on programming language
Improper Following of Specification by Caller, CWE-573	Object Model Violation: Just One of Equals and Hashcode Defined, CWE-581	Redefinition of either Equals or Hashcode without a pair	Java
Error Conditions, Return values, Status Codes, CWE-389	Return Inside Finally Block, CWE-584	Return Inside the finally block	Java

3.1.1. Example of the Code Weakness in Random Value Prediction

Let us consider a CWE-337 weakness determined under CWE as a predictable seed in PRNG. If a seed used in the program for PRNG is predictable (*i.e.* formed on the basis of system time, process identifier and other easily predictable parameters), any attacker may guess values generated by PRNG to a high degree of certainty. Depending on further use of the same, this may enable attacks at session keys, password hashes, coding gamma sequence, temporary files substitution, etc.

To detect defects, the following heuristic algorithm can be created :

1. Search for functions responsible for setting a seed in PRNG;
2. Analysis of the seed to verify whether any constant and/or statistically predictable value (such as a week day) is used for its initialization regarding that such values could have been determined earlier;
3. If the seed is formed on the basis of these data only it is likely that there is a CWE-337 weakness in software.

The production model of such weakness (CWE-337) is as follows :

$$\begin{cases} \mathrm{R}_1^{337} = < s_j \in S \; ; s_j \in o_{asg}(s_{j-1}, \mathrm{Y}) \, ; (\mathrm{Y} \cap \mathrm{I}_{\det} \neq \varnothing) \rightarrow \mathrm{AP}\,(\mathrm{s}_{j-1,}\mathrm{I}_{\det}) > \\ \quad \mathrm{R}_1^{337} = < s_j \in S \; ; s_j \in \mathrm{M}_{func} \wedge z \in \mathrm{F}_{\mathrm{prng}} \; ; \; f_{arg} \cap \mathrm{I}_{\det} \neq \varnothing) \rightarrow \\ \qquad\qquad \mathrm{AP}\,(\mathrm{s}_{j+1,}\mathrm{L}); \mathrm{P}_2(337, 6) > \end{cases} \tag{6}$$

Let us consider the first rule R_1^337 of the built production model. The software code area where a potentially unsafe construction is most likely to appear is any operator (function) which assigns any value of the set of information objects Y, to the operand *x*, *i.e*. CP_1 (S, N, M, F, *o*, D, L) = CP_1 (*o*) = $\mathrm{N}_{ij} \in o_{asg}$ (*x*,Y).

The core of the first heuristic production under consideration is as follows: "**if** any information object is assigned one of statistically predictable values, then this information object has a statistically predictable value".

Let us consider the second rule R_2^{337} of the built production model. The software code area where a potentially unsafe construction is most likely to appear is any operator (function) which transfers the control to the functional object, in this case it is any operator (function) generating API which initializes a pseudorandom numbers generator.

The core of the second heuristic production under consideration is as follows: "if API which initializes a pseudorandom number generator has predictable arguments, it is a potentially unsafe construction".

The production rule post-condition implies conclusions on a potentially unsafe construction (weakness) numbered 337.

3.1.2. Example of the Code Weakness Such as the Object Model Violation

Let us consider the code weakness CWE-581, namely the object model violation. The weakness is that there is only one of Equals or hashCode defined. This is due to that Java implies fixed properties of objects related to the check of their identities. One of such properties is that identical objects should have equal hashes. In other words, the check should be as follows:

if a.equals(b) == true, then a. hash Code() == b. hash Code ().

If only one of such methods is defined, identical objects will no longer be equal, and vice versa, in some cases different objects may become equal. This may result in errors in collections, maps and sets.

Description of the heuristic algorithm for defect detection :

1. Search in classes of Equals redefinition;
2. Search in classes of hashCode redefinition;
3. Warning of the possible code weakness CWE-581 if there is only one of them.

The production model of the heuristic rule is as follows:

$$\begin{cases} \mathrm{R}_1 = < s_j \in \mathrm{S} \, ; s_j \in \mathrm{M}_{\mathrm{obj}} \wedge t_{obj}\,(s_j) = \text{finally} \, ; (f_{\mathrm{inside}}(s_j) \cap \mathrm{M}_{\mathrm{info}} \cap \{\text{equals, hash Code}\} \neq \varnothing \\ \qquad\qquad\qquad \rightarrow \mathrm{AP}\,(s_j, \mathrm{L}) \, ; \mathrm{P}_1(581, 3) > \end{cases}$$

3.1.3. Example of the Code Weakness Such as Incorrect Value Return

Let us consider the code weakness CWE-584, namely incorrect Return Inside Finally Block. The weakness is as follows: Return Inside Finally Block rejects all exceptions generated in the Try block, thus emergencies are processed, including emergencies related to security mechanisms.

Description of the heuristic algorithm for weakness detection :

1. Search for Finally blocks;
2. Analysis of the block bodies to check whether return statements are present.

The production model CWE-584 can be as follows:

$$\{R_1 = < s_j \in M_{\text{func}} ; t_{obj}(s_j) = \text{'finally'} ; I_{\text{ret}} \in f_{\text{inside}}(s_j) \to AP(s_j, L) ; P_1(584, 2) > \qquad (8)$$

4. DISCUSSION

Efficiency and Productivity of the Heuristic Analysis

The proposed method for detection of a wide range of vulnerabilities by the heuristic analysis has applied significance, as it is already used in practice and received approval in real projects (more than 500 code audits under the certification testing of software products of leading manufacturers such as Microsoft, Huawei, Symantec, McAfee, Kaspersky).

The developed theoretical contents have become a basis for creating AppChecker implemented on the basis of PHP, Java and C/C++ parsers. The production rule of the analyzer is fulfilled by XPath-queries for XML-view of the abstract syntax tree of the code.

Check completeness is ensured by reference to the international register of security weaknesses, CWE. Currently the analyzer (version 2.1.10) includes 99 heuristics for Java, 92, for C/C++ and 32, for PHP.

The comparison analysis of static analyzers (based on checks of properties and search by templates) using test examples with an open code showed their similar productivity and performance. Results of the comparison analysis of synthetic (test) data sets according to SATEC are shown in (Markov, Fadin, Shvets and Tsirlov, 2015).

It should be understood that heuristic approaches depend mostly on experts' qualifications both in heuristics development and analysis of a potentially unsafe code fragments. At the same time, deliberate software bugs may be identified only by the heuristic analysis.

The proposed method is not the only one: in fact, a synthesis of possible methods for detection of a certain class of weaknesses by response rate and I and II type errors is important [24,25]. For example, ISO/IEC TR 20004 proposes methods for detection of known vulnerabilities published in open security bulletins. Such approach increases efficiency of structural testing, for example, during the initial analysis of appropriate components.

5. CONCLUSION

The following conclusions have been made on the basis of the research results:

1. The proposed heuristic method ensures completeness and consistency of checks, as it does not restrict classifications of weaknesses in the software code;
2. The approach can be used as a basis for practical combination of different methods for static analysis of security software (with regard to opinions of qualified experts);
3. The specifics of the heuristic method are those that it may identify deliberate bugs by expert features of a destructive code;
4. The developed metamodel allows for formal description of weaknesses search heuristics at any level of the code presentation (for example, abstract syntax tree and abstract semantic graph), which improves the unification of complex vulnerabilities detection. The metamodel advantages include:
    - visualization and relative simplicity of processing heuristic algorithms for vulnerabilities detection, and template (signature) base itself;
    - high operating speed;
    - easy signatures modification and porting to different platforms and software languages.

The proposed method and tools are useful not only for accredited testing laboratories but for secure software engineers.

6. REFERENCES

1. Ayewah, N., Hovemeyer, D., Morgenthaler, J., Penix, J. and Pugh, W. (2008). Using Static Analysis to Find Bugs (vol. 25, no. 5, pp. 22-29). IEEE Softw.
2. Baier, C. and Katoen, J. (2008). Principles of model checking. Cambridge, Mass.: MIT Press.
3. Boulanger, J. (2012). Static analysis of software. London, UK: ISTE/Wiley.
4. Bronshteyn, I. (2013). Study of defects in a program code in Python (vol. 39, no. 6, pp. 279-284). Programming and Computer Software.
5. Chen, H., Wagner, D. (2002). MOPS: an infrastructure for examining security properties of software (pp. 235-244). In 9th ACM conference on Computer and communications security. CCS'02, ACM.
6. Glukhikh, M., Itsykson, V., Tsesko, V. (2012). Using Dependencies to Improve Precision of Code Analysis (vol. 46, no 7, pp. 338-344). Automatic Control and Computer Sciences.
7. Ivannikov, V.P., Belevantsev, A.A., Borodin, A.E., Ignatiev, V.N., Zhurikhin, D.M. and Avetisyan, A.I. (2014). Static Analyzer SVACE for Finding Defects in a Source Program Code (vol. 40(5), pp. 265-275). Programming and Computer Software.
8. Hovemeyer, D., Spacco, J. and Pugh, W. (2006). Evaluating and tuning a static analysis to find null pointer bugs (vol. 31, no. 1, p. 13). SIGSOFT Softw. Eng. Notes.
9. Logozzo, F. and Fähndrich, ?. (2008). On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis (4959, pp. 197-212). LNCS.
10. Seo, S., Gupta, A., Mohamed Sallam, A. Bertino, E. and Yim, K. (2014). Detecting mobile malware threats to homeland security through static analysis (vol. 38, pp. 43-53). Journal of Network and Computer Applications.
11. Zhu, F. and Wei, J. (2014). Static analysis based invariant detection for commodity operating systems (vol. 43, pp. 49-63). Computers & Security.
12. Barabanov, A., Markov, A., Fadin, A. and Tsirlov, V. (2015). A Production Model System for Detecting Vulnerabilities in the Software Source Code (pp. 98-99). In Proc. 8th Int. Conf. on Security of Information and Networks, Sochi, Russian Federation.
13. Static Analysis Technologies Evaluation Criteria v1.0, Web Application Security Consortium, 2013. URL: http://projects.webappsec.org/w/file/fetch/66107997/SATEC_Manual-02.pdf.
14. Muske, T. and Bokil, P. (2015). On implementational variations in static analysis tools (pp. 512-515). IEEE 22nd Int. Conf. on Software Analysis, Evolution, and Reengineering, Montréal, Québec, Canada.
15. Chess, B. and West, J. (2007). Secure programming with static analysis. Upper Saddle River, NJ: Addison-Wesley.
16. AlBreiki, H. and Mahmoud, Q. (2014). Evaluation of static analysis tools for software security (pp. 93-98). In 10th Int. Conf. on. Innovations in Information Technology, Al Ain, UAE.
17. Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R. (2013). Why Don't Software Developers Use Static Analysis Tools to Find Bugs? (pp. 672-681). Int. Conf. Softw. Eng.
18. Markov, A., Fadin, A., Shvets, V. and Tsirlov, V. (2015). The experience of comparison of static security code analyzers (vol. 5(3), p. 55). International Journal of Advanced Studies, IJAS.
19. Kulenovic, M. and Donko, D. (2014). A survey of static code analysis methods for security vulnerabilities detection (pp. 1381-1386). In 37th Int. Conv. on Information and Communication Technology, Electronics and Microelectronics, Opatija, Croatia.
20. McLean, R. (2012). Comparing Static Security Analysis Tools Using Open Source Software (pp. 68-74). IEEE 6th Int. Conf. on Software Security and Reliability Companion, Washington, D. C., USA.
21. Xin, L. and Wandong, C. (2011). A program vulnerabilities detection frame by static code analysis and model checking (pp.130-134). IEEE 3rd International Conference on Communication Software and Networks, Xi'an, China.
22. Brat, G., Navas, J. A., Shi, N. and Venet, A. (2015). IKOS: A framework for static analysis based on abstract interpretation (pp. 271-277). In 13th Int. Conf. on Software Engineering and Formal Methods, York, UK.
23. Panichella, S., Arnaoudova, V., Di Penta, M. and Antoniol, G. (2015). Would static analysis tools help developers with code reviews? (pp. 161-170). IEEE 22nd Int. Conf. on Software Analysis, Evolution, and Reengineering, Montréal, Québec, Canada.
24. Barabanov, A., Markov, A. and Tsirlov, V. (2016). Methodological Framework for Analysis and Synthesis of a Set of Secure Software Development Controls (vol. 88(1), pp.77-88.). Journal of Theoretical and Applied Information Technology.
25. Howard, H., Lipner, S. (2006). The Security Development Lifecycle: A Process for Developing Demonstrably More Secure Software. Microsoft Press.