Making Static Code Analysis More Efficient

Pomorova O.V. and Ivanchyshyn D.O.

System Programming Department, Khmelnytskyi National University, Instytutska Str. 11, Khmelnytskyi, 29016, Ukraine, E-mail: o.pomorova@gmail.com, dmytro_ivanchyshyn@ukr.net

Received 19 March 2014; Accepted 27 April 2014;
Publication 2 June 2014

Abstract

Modern software is a complex high-tech product. Users and customers put forward a number of requirements to such products. Requirements depend on software purpose. However, reliability, fault tolerance, security and safety requirements are topical for all software types. One of the approaches for realization of such requirements in the implementation stage of software life cycle is a static source code analysis (SCA). The efficiency assessment task of the SCA tools is an actual problem. This paper presents the method of the efficiency evaluating of the software static source code analysis. It allows increasing the quality and reliability of software in general. The result of this work is a method of efficiency improving at the debugging stage and approach for selection of the static code analysis tools for software of various types.

Keywords: Source code analysis, security, vulnerabilities, weaknesses, static analysis efficiency, efficiency metrics.

1 Introduction

The company Veracode provides a "cloud" service for the analysis of software vulnerabilities. Every year it presents a detailed analysis of the vulnerabilities, which were discovered in software. Veracode State of Software Security (SoSS) Report Volume 5 examines data collected over an 18 month period from January 2011 through June 2012 from 22,430 applications [1]. This report

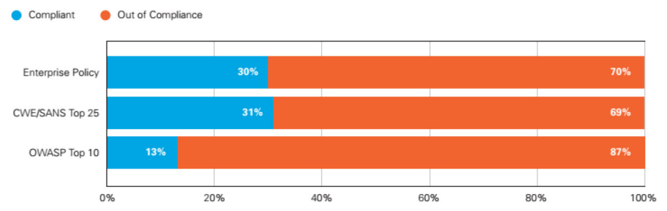


Figure 1 Compliance with Security Policies upon First Submission

examines application security quality, remediation, and policy compliance statistics and trends. It shows that 70% of applications failed to comply with enterprise security policies on first submission. Web applications are assessed against the Open Web Application Security Project (OWASP) Top 10 and only 13% complied on first submission. Non-web applications are assessed against the Common Weakness Enumeration (CWE/SANS) Top 25 and 31% complied on first submission. Only 30% of applications complied with enterprise defined policies (Figure 1). Therefore, the vast majority of applications are returned to the testing stage for further debugging.

Detection and correction of software defects (debugging) are two of the most difficult and time-consuming stages in the process of software development. Up to the 95% of debugging time is spent on the detection of defects and only 5 % is spent for defect correction [2]. So one of the actual issues for today is improving of the efficiency of software error detection. Reducing the time for identification of defects in software source code will significantly decrease the general time and resources that are spent in the debugging stage of the software lifecycle.

One advantage of our approach is the ability to consider and to identify actual vulnerability to a particular type of software at the implementation stage. It allows us to increase the quality and reliability of software in general. This paper presents an analysis of the weaknesses classification and identification problems and describes the ability to apply such information in the realization stage of the software lifecycle. The result of this work is an efficient method of improving the debugging stage and an approach for the selection of static code analysis tools for software of various types.

2 Static Code Analysis (SCA)

One of the methods of source code verification is static code analysis. SCA is the process of evaluating a system or component based on its form, structure, content, or documentation [3]. From a software assurance perspective, static analysis addresses weaknesses in program code that might lead to vulnerabilities. SCA is performed without actually executing programs and can be applied on the early stages of software lifecycle. Such analysis may be manual, as in code inspections or automated through the use one or more tools. Automated static code analyzers typically check source code but there is a smaller set of source code analyzers that check byte code and binary code. There are especially useful when source code in not available. Static code analyzers are used to uncover hard to find implementation errors before run-time, since they may be even more difficult or impossible to find and assess during execution. These tools can discover many logical, safety and security errors in an application without the need to execute the application.

2.1 Static Security Analysis (SSA)

One of the SCA categories is static security analysis (SSA). It attempts to identify errors and security weaknesses through deep analysis of source code [4]. SSA is primarily aimed at developers and QA engineers who wish detect software defects early in the development cycle in order to reduce time and cost. It also assists developers in hardening their application against security attack. SSA provides an effective way to discover defects, especially in code that is hard to exercise thoroughly with tests.

Many coding errors and patterns of unsafe usage can be discovered through static analysis. The main advantage of static analysis over dynamic analysis is that it examines all possible execution paths and variable values, not just those that are provoked during testing. This aspect of static analysis is especially valuable in security assurance, since security attacks often exercise an application in unforeseen and untested ways. SSA can detect different error conditions: buffer overflows and boundary violations, misuse of pointers and heap storage, memory leaks, use of uninitialized variables and objects, unsafe/incorrect use of functions, etc.

2.2 Static Code Analysis Efficiency

Static code analysis efficiency is a complex property that reflects the quality of the results, the degree of automation of the analysis and the complexity of its organization, resource-intensive, applicability to different class and size of programs [5]. The obtained results (based on the number of true positive (TP), false positive (FP), and false negative (FN) in the analyzer's reports) (2.1, 2.2). Other parameters are the importance of revealed defects, the properties of programming languages, type of software, features of the software algorithm and coding style, the impact of environment and external influences.

The F_score metric provides weighted guidance in identifying the most efficient static analysis tool by capturing how many of the weaknesses were found (true positives) and how much noise (false positives) was produced. An F_score is harmonic mean of the Precision and Recall values (2.3):

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F\_Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3}$$

These parameters are universal and can be applied both for SCA and for SSA efficiency assessment.

2.3 Static Security Analysis Tools

SSA has a number of usage features. The security Department of the U.S. National Institute of Standards and Technology provides a list of SCA tools that can be used for detecting and reporting weaknesses that can lead to security vulnerabilities. The following SCA tools were select for this paper: Gimpel PC Lint ($389), PVS – Studio ($4585), Red Lizard Goanna Studio ($999), and CppCheck (freeware).

Three sets of test samples for quality of the SCA assessment were selected. The first set was taken from the website of the U.S. department of Homeland Security. The second set was taken from the website of the Security Department of U.S. National Institute of Standards and Technology. They included test samples designed specifically for SCA tools testing [6]. The examples are small, simple C/C++ programs, each of which is meant to evaluate some specific aspect of a security scanner's performance. The third test set contained applications with only one type of defect. The widespread defect in the source code CWE (Common Weakness Enumeration) 476 CWE null pointer dereference was selected. Each of the tools has a rule for identifying of such class of defects. The third test set is aimed at detection the differences between the methods used by developers to identify the stated defects. Table 1 shows the F _score metric results obtained for the three test sets.

Table 1 SCA Results

F_score	CppCheck	PVS-Studio	Goanna	PC-Lint
Test set 1	0,74	0,39	0,21	0,44
Test set 2	0,31	0,31	0,35	0,33
Test set 3	0,75	0,67	0,67	0,67

The efficiency of SCA tools varies significantly and changes greatly for different test sets. In addition, the list of defects detected by each analyzer was different. Modern investigations of other SCA tools efficiency also shows similar results [7, 8]. Therefore:

The efficiency of the tools depends on the usage scenario
SCA tools don't identify all defects in existing software

Consequently, it is hard for software developers and QA managers to choose one of the available SCA tools. In addition, tools have to be up to date with respect to the spectrum of threats, weaknesses, vulnerabilities and long-term costs.

2.4 Improving of the SCA Efficiency

Today there are a number of investigations on improving the efficiency of SCA. There are a number of regulations to ensure the quality of the static analysis of the application in the development of military software. Michael Howard proposes a list of recommendations for improving SCA efficiency [9]:

Multiple tools should be used to offset tool biases and minimize false positives and false negatives and minimize false positives and false negatives.
Analysts should pay attention to every warning and error.
Warnings from multiple tools may indicate that the code needs closer scrutiny (e.g. manual analysis).
Code should be evaluated early, preferable with each build, and re-evaluated at every milestone.

In addition, analysts should make sure that code reviews cover the most common vulnerabilities and weaknesses, such as integer arithmetic issues, buffer overruns, SQL injection, and cross-site scripting (XSS). Sources for such common vulnerabilities and weaknesses include the Common Vulnerabilities and Exposures (CVE) and Common Weaknesses Enumeration (CWE) databases, maintained by the MITRE Corporation. MITRE, in cooperation with the SANS Institute, also maintains a list of the "Top25 Most Dangerous Programming Errors" that can lead to serious vulnerabilities. Static code analysis tool and manual techniques should at a minimum address this Top 25. The better static code analysis tools are expensive. Other ways to improving the efficiency of SCA is to use multiple tools to offset tool biases and minimize false positives and false negatives. However, this is cost prohibitive.

The practical implementation of the recommended steps has a number of problems. Different SCA tools has different defects classification systems. It is hard to compare results obtained from different tools. Also, the bases of some analyzers do not cover the most common vulnerabilities and weaknesses. All of which make it impossible to fully use all benefits of the static analysis technology.

3 An Efficient Method for Static Code Analysis

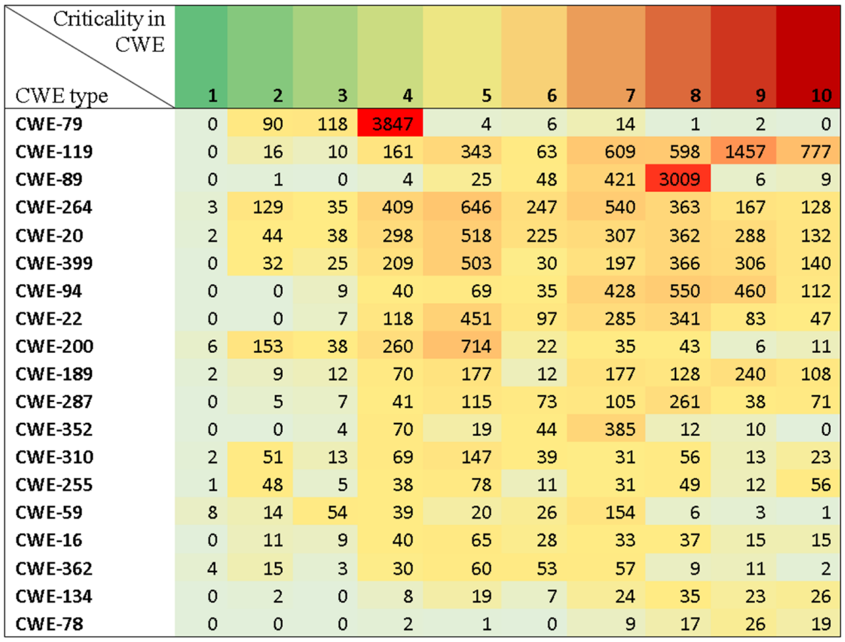
3.1 Problems with Weaknesses Classifications

Every few years a «Top 25 Most Dangerous Software Errors» is constructed from the CWE. This is a list of the most widespread and critical errors that can lead to serious vulnerabilities in software. The higher a defect is located in the CWE ranking, the more important its detection by SCA tools. However, the rating is developed for all existing types of software, which complicates its practical application. For example, currently the most important is weaknesses in the list is CWE-89 (cross-site scripting). This problem is related to web services and it is not relevant to application or system software. Thus, for a particular type of software it is appropriate to apply special an importance rating of the defects (Table 2).

One approach to solving this problem is to carry out an analysis of NVD (national vulnerabilities database). The task was to identify the most common and critical defects for certain categories of software. NVD is the U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol (SCAP). NVD includes databases of security checklists, security related software flaws, misconfigurations, product names, and impact metrics. Figure 2 shows the number and criticality of all weakness obtained from NVD database. Table 2 presents weaknesses ratings with different criteria: relevance, most important weaknesses for all types of software, relevance only for operating systems.

Table 2 The Most Important Weaknesses

No	Top CWE 2011	Relevance	Most important	Relevance in OS
1	CWE-89	CWE-79	CWE-119	CWE-399
2	CWE-78	CWE-119	CWE-89	CWE-20
3	CWE-120	CWE-89	CWE-264	CWE-119
4	CWE-79	CWE-264	CWE-79	CWE-264
5	CWE-306	CWE-20	CWE-20	CWE-189
6	CWE-862	CWE-399	CWE-94	CWE-200
7	CWE-798	CWE-94	CWE-399	CWE-362
8	CWE-311	CWE-22	CWE-22	CWE-94
9	CWE-434	CWE-200	CWE-189	CWE-16
10	CWE-807	CWE-189	CWE-200	CWE-310
11	CWE-250	CWE-287	CWE-287	CWE-287
12	CWE-352	CWE-352	CWE-352	CWE-79
13	CWE-22	CWE-310	CWE-310	CWE-255
14	CWE-494	CWE-255	CWE-255	CWE-22



Criticality in CWE / CWE type	1	2	3	4	5	6	7	8	9	10
CWE-79	0	90	118	3847	4	6	14	1	2	0
CWE-119	0	16	10	161	343	63	609	598	1457	777
CWE-89	0	1	0	4	25	48	421	3009	6	9
CWE-264	3	129	35	409	646	247	540	363	167	128
CWE-20	2	44	38	298	518	225	307	362	288	132
CWE-399	0	32	25	209	503	30	197	366	306	140
CWE-94	0	0	9	40	69	35	428	550	460	112
CWE-22	0	0	7	118	451	97	285	341	83	47
CWE-200	6	153	38	260	714	22	35	43	6	11
CWE-189	2	9	12	70	177	12	177	128	240	108
CWE-287	0	5	7	41	115	73	105	261	38	71
CWE-352	0	0	4	70	19	44	385	12	10	0
CWE-310	2	51	13	69	147	39	31	56	13	23
CWE-255	1	48	5	38	78	11	31	49	12	56
CWE-59	8	14	54	39	20	26	154	6	3	1
CWE-16	0	11	9	40	65	28	33	37	15	15
CWE-362	4	15	3	30	60	53	57	9	11	2
CWE-134	0	2	0	8	19	7	24	35	23	26
CWE-78	0	0	0	2	1	0	9	17	26	19

Figure 2 Number and Criticality of Weaknesses from NVD Database

For operating systems, the distribution of weaknesses varies depending on the operating system: Windows, RedHat, Novel, Solaris, Apple or others (Figure 3).

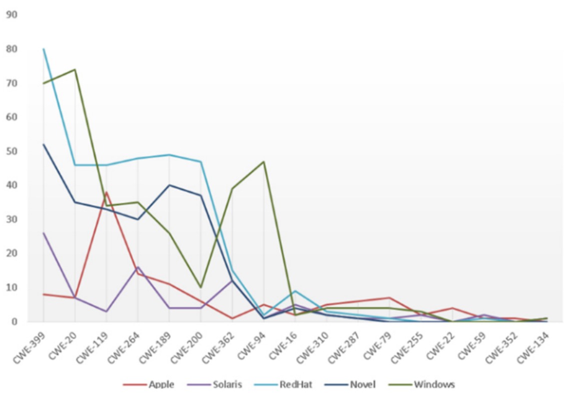


Figure 3 Distribution of Weaknesses for OSs

Figure 3 shows that for different types of software the distribution of defects varies significantly. It is reasonable to use different SCA tools for particular types of software.

3.2 Improving the Efficiency of SCA

The first step is to determine the type of software (ST) (Figure 4). The analysis of the NVD database shows that for different types of software there is a different list of most widespread weaknesses.

The second step, the informational sources (IST) are chosen. From the set of sources that contain information about the defects identified in the software, the most appropriate sources for subject area are selected. For example, for SSA such source can be the NVD.

Development of the defects list for a given type of software (STw) is the third step. Based on the analysis of the prevalence and criticality of defects given in IST the rating of the most important of defects is formed. The list includes weaknesses that have to be identified by SCA tools.

The fourth step is formation of the test samples set (TST) that cover the list of weaknesses (STw). For each weakness the test samples set that will be used for SCA checking is formed. For example, for buffer overflow weakness CWE-120 a set of test samples are given in Table 3.

The fifth step is the static analysis of test samples set by SCA tools. The result of the stage is a log-file with defects that have been detected.

Parsing of the log-files is the sixth step. The number of false positive, false negative, and true positive, is calculating in this stage.

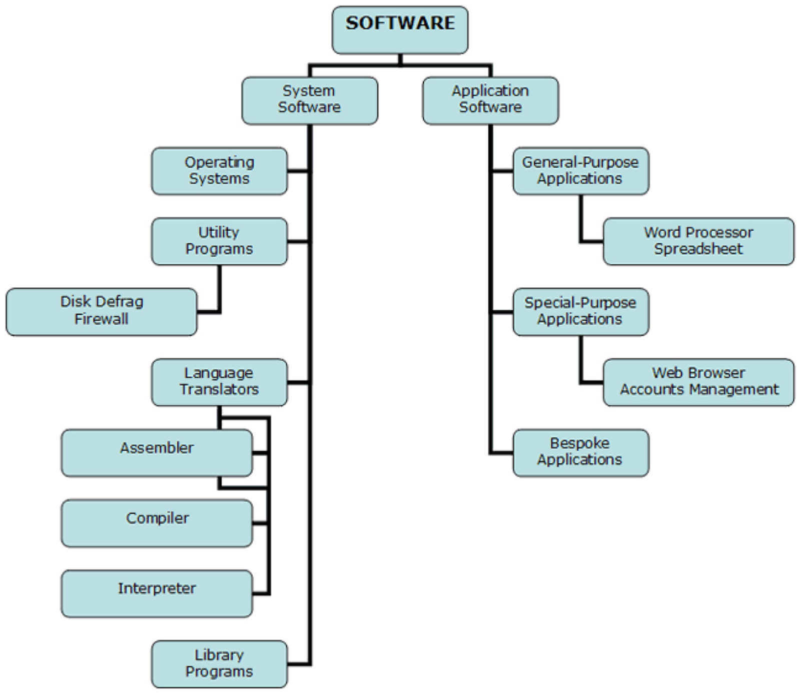


Figure 4 Types of software

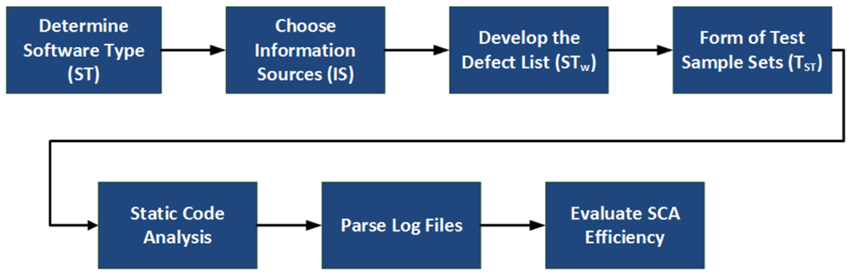

Determine Software Type (ST)
Choose Information Sources (IS)
Develop the Defect List (ST_w)
Form of Test Sample Sets (T_ST)
Static Code Analysis
Parse Log Files
Evaluate SCA Efficiency

Figure 5 Method of the Static Code Analysis Efficiency Assessment

The last step is an analysis of the efficiency of the SCA. This step includes the calculation of Precision, Recall and F_score metrics. They are based on the number of true positives (TP), false positives (FP), and false negatives (FN) in the analyzer's report. The F_score metric is a final result that can be used for different SCA tools comparison. Figure 5 presents the steps for our method.

Table 3 Set of tests for buffer overflow weakness

Error type	Error name	CWE number
Arrays	Direct overflow	CWE-119
	Off-by-one errors	CWE-193
	Unbounded copy	CWE-120
Strings	Direct overflow	CWE-119
	Null termination	CWE-170
	Off-by-one errors	CWE-193
	Truncation error	CWE-222
	Unbounded copy	CWE-120
	Strcpy_check	
	strcat_check	
	gets_check	
	strncpy_check	
	strncat_check	
	fgets_check	
Integer	Overflow	CWE-190
	Sign errors	CWE-195
	Truncation errors	CWE-197

The user choses only the type of software and the information sources. The result is the most appropriate static code analysis tool for concrete software project.

4 Conclusion

Our investigation shows the existence of significant differences in the results produced by static code analysis tools. The efficiency of modern SCA tools depends on the usage scenario. In terms of software security, it depends on consideration of information about actual vulnerabilities. Such data can be obtained from databases like the NVD. Therefore, it is advisable to integrate information about known vulnerabilities and weaknesses in source code into the code analysis process.

Consequently, for software quality improvement and security assurance it is necessary to pay attention to the problem of the choosing appropriate SCA tools. The efficiency of a tool has to be evaluated and the information about actual vulnerabilities ought to be taken into account even at the software lifecycle stage of implementation. The proposed method provides a solution to this problem.

The problems of informational sources choosing for particular type of software, test sets development for defects checking and automatic parsing of log files are needed further investigations.

References

[1] Veracode Inc., State of Software Security Report: Volume 5, April 2013, 44 p.

[2] Ian Sommerville, Software Engineering (9th Edition), 2010.

[3] R. Lopes, D. Vicente, N. Silva. Static Analysis tools, a practical approach for safety-critical software verification. Critical Software SA Parque Industrial de Taveiro. Coimbra, Portugal, 2009, 12 p.

[4] Intel Corporation, Improve C++ Code Quality with Static Security Analysis (SSA), 2013, 11 p.

[5] National Security Agency Center for Assured Software. On Analyzing Static Analysis Tools. July, 2011.

[6] Build Security In. Source Code Analysis Tools - Example Programs: https://buildsecurityin.us-cert.gov/bsi/articles/tools/code/498-BSI.html

[7] Thomas Hofer. Evaluating Static Source Code Analysis Tools, School of Computer and Communications Science, Ecole Polytechnique Federal de Lausanne, March 12, 2010

[8] R. Plösch, A. Mayr, G. Pomberger, M. Saft. An Approach for a Method and a Tool Supporting the Evaluation of the Quality of Static Code Analysis Tools. Proceedings of SQMB 2009 Workshop, SE 2009 conference, Kaiserslautern, Germany, July 2009.

[9] Howard, M. A Process for Performing Security Code Reviews, IEEE Security & Privacy, July-August 2006, pp. 74–79.

Biographies

Oksana Pomorova. Doctor of Technical Science, Head of System Programming Department, Full Professor in Khmelnitsky National University (Ukraine). Received the PhD degree in Kyiv Institute of Automatics (2002), the degree Doctor of Technical Science in 2008 in the National University "Lviv Polytechnic" (Ukraine), specialty 05.13.13 - "Computers, Systems and Networks". IEEE member from 2005. *Teaching* - Computer Modeling, Technology of Software Design, Artificial Intelligence Systems. *Guest lectures*: Department of Computer Systems and Networks, Yuriy Fedkovych Chernivtsi National University (Ukraine); Kielce University of Technology (Poland). *Research Interests*: Intelligent Methods and Means of Computer Systems Diagnosing, Quality Assessment of Critical Software; Modeling and Design of Knowledge Bases for Testing and Diagnosing Specialized Computer Systems.

Dmytro Ivanchyshyn. He defended his master's thesis in Khmelnitsky National University. Now studying in postgraduate at the Faculty Programming, Computer and Telecommunication Systems and working as teacher trainee of System Programming Department. His research interests are: Software Quality Assurance and Testing, System Security