Evaluation of Static Analysis Tools for Software Security

Hamda Hasan AlBreiki
Department of Computer Information Science
Higher Colleges of Technology, UAE

Qusay H. Mahmoud
Department of Electrical, Computer and Software Eng.
University of Ontario Institute of Technology, Canada

***Abstract*—Security has been always treated as an add-on feature in the software development lifecycle, and addressed by security professionals using firewalls, proxies, intrusion prevention systems, antivirus and platform security. Software is at the root of all common computer security problems, and hence hackers don't create security holes, but rather exploit them. Security holes in software applications are the result of bad design and implementation of software systems and applications. To address this problem, several initiatives for integrating security in the software development lifecycle have been proposed, along with tools to support a security-centric software development lifecycle. This paper introduces a framework for evaluating security static analysis tools such as source code analyzers, and offers evaluation of non-commercial static analysis tools such as Yasca, CAT.NET, and FindBugs. In order to evaluate the effectiveness of such tools, common software weaknesses are defined based on CWE/SANS Top 25, OWASP Top Ten and NIST source code weaknesses. The evaluation methodology is based on the NIST Software Assurance Metrics And Tool Evaluation (SAMATE). Results show that security static analysis tools are, to some extent, effective in detecting security holes in source code; source code analyzers are able to detect more weaknesses than bytecode and binary code scanners; and while tools can assist the development team in security code review activities, they are not enough to uncover all common weaknesses in software. The new test cases developed for this research have been contributed to the NIST Software Assurance Reference Dataset (samate.nist.gov/SARD).**

Keywords—software security; static analysis; *OWASP; SAMATE; security metrics.*

I. Introduction

Software applications have become an integral part of our daily life, as they're used in manipulating and holding our personal information, health records, financial transactions, business data and other sensitive information. The questions that should come to mind include: to what extend can we trust software? And who is responsible for ensuring the security of these software applications we are depending on, especially when they operate in the cloud? This section presents an overview of software security and a brief history of it, followed by a definition of the problem statement, and an outline of the organization of this paper.

Securing software systems has always been addressed by network security people in the perimeter security using firewalls, proxies, intrusion prevention systems, anti-virus and platform security tools [1]. Security needs to be addressed at all levels of the defense stack, including: network, host, application and data [2]. Software is at the root of all common computer security problems, because hackers don't create security holes, but simply exploit them. These holes resulted from bad design and implementation of software [3]. It is estimated that 90 percent of reported security incidents result from exploits against defects in the design or code of software [4]. Security needs to be addressed from another angle; it needs to be built within the application [1, 2, 5, 6].

In the last ten years, several initiatives have been introduced for integrating security in the software development lifecycle. All these initiatives added new activities to the different phases of the software development lifecycle in order to enhance security of software products. Threat modeling and security code review are two of security activities that were included in most software security initiatives. Today, many tools are available to support security code review activities, including security static analysis tools that are automated tools that analyze programs to detect security weaknesses without executing the code. The analysis can be done at the source code level, the bytecode level, or the binary code level.

In the mid 1990s, there was nothing written about software security [6, 7]. McGraw and Ed Felton found major holes in Java at the time and published the "Java Security" book in August 1996. Later in 2001, the first book on software security was written: "Building Secure Software" by John Viega and Gary McGraw [7, 8], and the idea of secure software engineering started to emerge and grow. Viega and McGraw re-presented data collected by David Lake between 1998 and September 2000, which illustrated that approximately 20 new vulnerabilities in software were made public each week, and many of these vulnerabilities existed for long time before they were discovered [3]. The Internet and the Web created a great threat environment full of risks for all software users in this interconnected world [9, 10].

On January 15, 2002, an email from Bill Gates was sent to Microsoft employees that introduced a concept known as Trustworthy Computing to Microsoft and the whole IT industry. Bill Gates stated: "what I believe is the highest priority for the company and for our industry over the next decade: building a Trustworthy Computing environment for customers that is as reliable as the electricity that powers our homes and businesses today." [11]. This new concept changed the way Microsoft is developing software and it has been leading the whole IT industry to a complete new level of trustworthiness in computing [12]. The challenge at the time was that there were no best practices to rely on [13].

Software security is the idea of engineering software so that it continues to function correctly under a malicious attack [14]. Another definition for software security is “the ability of software to resist, tolerate, and recover from events that intentionally threaten its dependability.” [15]. Lipner [16] considered security as a core requirement for software venders to meet the demand for protecting critical infrastructures and build and preserve the widespread trust in computing. Having excellent software developers and architects will not produce secure products. Most developers and architects have not had the opportunity to learn much about security and universities’ security courses emphasize network security [3]. Many types of successful attacks on software applications were caused by developers’ lack of attention to security in the design, development, deployment and maintenance of their products [5]. Many people consider security as an operating system problem or a network perimeter and firewall problem, which is not the case all the time [9]. For this reason security was always a non-functional requirement [1]. The failure of software developers to take security view of their products from inception through deployment and beyond, software is at the root of most security and privacy breaches [5]. Therefore the demand for software security specialists has grown dramatically and several governmental, commercial, and educational organizations are all interested in advancing the software security workforce's educational preparation [17].

Like all other engineering disciplines, software engineering involves a structured sequence of stages to develop a software product. These stages are known as the software development lifecycle. The main stages, whether using traditional or agile methodologies, are: requirements analysis, design, implementation, testing, deployment and maintenance.

None of traditional methodologies used for software development lifecycle have considered security as a deliverable in any of the stages of the lifecycle. Security has been always treated as an add-on feature in software, which explains the reason behind security bugs and flaws that are exploited by hackers today.

To this end, this work introduces an evaluation framework for the effectiveness of security static analysis tools. These tools are used for security code review practices to support developers in integrating security into the software development lifecycle. It analyzes the strengths and weaknesses of the available non-commercial tools for security code review practices, and evaluates the effectiveness of such tools to uncover software security weaknesses.

The rest of this paper is organized as follows. Section II discusses the related work. Section III explores non-commercial static analysis tools used for security code review. Section IV explains the evaluation methodology used for conducting this research. Section V presents and analyzes the results obtained from the evaluation. Finally, conclusions and future work are discussed in section VI.

II. RELATED WORK

Software security is an active area of research, and the last decade witnessed valuable contributions from both industry and academia in that field. This section discusses the leading initiatives for integrating security in the software development lifecycle and the common practices shared by these initiatives. These initiatives helped to bring attention to the need to build security into the software development lifecycle [18]. This section also provides an overview of the tools used to support a security-centric software development lifecycle, and the leading methods for evaluating such tools. Finally, the contributions of this research are presented.

A. Secure Software Development Lifecycle Initiatives

Most software developers are not aware of software security, and for decades they have been focusing on implementing system functionalities to meet delivery deadlines, and patching the inevitable bugs when it’s time for the next release [19]. In order to change these vital traditional development habits, software vendors need to move to a more stringent software development process that focuses, to a greater extent, on security. Such a process is intended to minimize the number of security vulnerabilities in the design, coding, and documentation and to detect and remove those vulnerabilities as early in the development lifecycle as possible [16]. NIST released a special publication under the title: "Security Considerations in the Information System Development Lifecycle" that offers a framework for integrating security in the generic software development lifecycle [20]. In addition, many other leading IT research groups have recognized the need for security to be integrated into the software development lifecycle [5].

In January 2002, Trustworthy Computing (TwC) was the starting point for Microsoft where development groups instigated "security pushes" to find ways to improve the security of existing code. By 2004 Microsoft Security Development Lifecycle (SDL) became a mandatory policy for software development at Microsoft [21]. Microsoft SDL involves modifying a software development organization’s processes by integrating measures that lead to improved software security. Figure 1 shows the key activities and where they should be integrated in SDL [22]. More recently, Microsoft published a whitepaper “Security Development Lifecycle for Agile Development”, which defines a way to embrace lightweight software security practices when using Agile software development methods. The document aims to meld the proven Microsoft SDL with Agile methodologies in a way that maintains the principles of both the Agile methods and the SDL process [23].

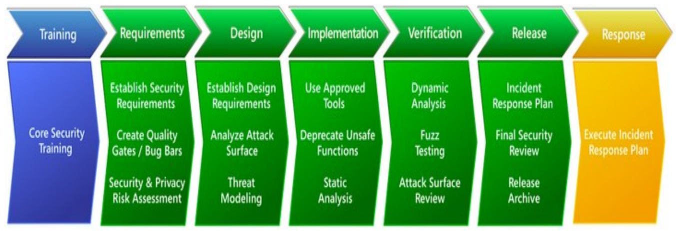


Figure 1: Microsoft SDL phases

Another high profile initiative that provides a well-organized and structured approach for moving security concerns into the early stages of the software development lifecycle is the Comprehensive, Lightweight Application Security Process (CLASP). It’s an open source application of Open Web

Application Security Project (OWASP) [24]. CLASP aims to give security a central role in the construction of software [25]. It integrates security processes into each phase of the development lifecycle and it defines 30 security activities and linked to one or more project role [26].

B. *Common Activities in Secure Software Development Lifecycle*

Secure software development is based on guidelines, best practices and undocumented expert knowledge. Current initiatives provide guidance for particular areas such as threat modeling, risk management, or secure coding [27, 28]. In 2008 McGraw and Chess defined a Software Security Framework (SSF) that aims to capture an overall high-level understanding that encompasses all of leading software security initiatives. They have identified a number of common domains and practices shared by most software security initiatives. SSF has twelve practices organized into four domains as shown in Table 1.

Table 1: SSF Domains and Practices [27]

Governance	Intelligence	SDL Touchpoints	Deployment
Strategy and Metrics	Attack Models	Architecture Analysis	Penetration Testing
Compliance and Policy	Security Features and Design	Code Review	Software Environment
Training	Standards and Requirements	Security Testing	Configuration Management and Vulnerability Management

The two most important software security practices from these twelve practices are architecture analysis and code review.

C. *Tools to support Security-Centric Software Development*

Today many tools are available to support a secure software development lifecycle. There are modeling and architecture tools for requirement and design phases, code analysis tools for implementation phase and black box testing and penetration testing tools for testing and deployment phases. All these tools aid the software development team in integrating security in the development lifecycle. White box tools are used for security code review practices, which is one of the most important practices used to integrate security in software development lifecycle. White box tools include static analysis tools and dynamic analysis tools. Source code security analyzers, bytecode scanners and binary code scanners are tools used for static analysis. These tools analyze software without executing the software. Source code security analyzer examines source code to detect and report weaknesses that can lead to security vulnerabilities. Bytecode scanners detect vulnerabilities in the bytecode and they are used in cases where the source code is not available. Binary code scanners detect vulnerabilities through disassembly and pattern recognition. These tools have the ability to look at the compiled result and factor in any vulnerability created by the compiler itself. On the other hand dynamic analysis tools analyze a running application for potential security vulnerabilities and can be used to complement static analysis tools in software lifecycle. These tools generate runtime vulnerability scenarios and analyze the software application from inside or outside. Web application vulnerability scanner is an example of a tool that analyzes the software from outside. This tool attacks the application over HTTP or HTTPS based on known vulnerabilities and attack patterns. Dynamic program analysis tools used to analyze the code inside an application. This kind of tools performs analysis in unmanaged languages and they perform memory debugging, memory leak detection, profiling and tracing [26].

D. *Tool Evaluation*

The tools and methods used for software security needs to be evaluated in order to assess their effectiveness and usefulness. One of the initiatives for evaluating software security tools is the NIST Software Assurance Metrics And Tool Evaluation (SAMATE). The NIST SAMATE project seeks to help develop standard evaluation measures and methods for software assurance [29]. It aims to improve software assurance by introducing methods for software tool evaluations, measuring the effectiveness of tools and techniques, and identifying gaps in tools and methods. One of the goals of SAMATE is to establish a methodology for evaluating software assurance tools. The NIST SAMATE project provides tool specification, test plans, and test sets.

One of the resources provided by NIST SAMATE project is Software Assurance Reference Dataset (SARD). SARD provides users, researchers, and software security assurance tool developers with a set of known security flaws. This allows end users to evaluate tools, and tool developers to test their methods. The dataset intends to encompass a wide variety of possible vulnerabilities, languages, platforms, and compilers. The dataset is anticipated to become a large-scale effort, gathering test cases from many contributors. Currently, the SARD contains test cases for C, C++, Java and PHP. In the current version of NIST special publication: Source Code Security Analysis Tool Test Plan, there is no test suites for the Java language. This publication offers three test suites for both C and C++.

The need for integrating security in the software development lifecycle has become a critical requirement for the software industry. Over the last decade, several initiatives for integrating security in the software development lifecycle have emerged, and tools have been developed to support these initiatives. The effectiveness of current approaches, methodologies and support tools need to be addressed and assessed.

III. NON-COMMERCIAL STATIC ANALYSIS TOOLS

Static analysis tools are used for code review activities in a security-centric software development lifecycle. These tools automatically analyze programs without executing the code. The analysis can be done at the source code level, bytecode level or binary code level. With static analysis no need to think about issues related to program executions such as the reachability of vulnerable code and the generation of input test cases to traverse the vulnerable code [30]. This research focuses on evaluating non-commercial static analysis tools. This section discusses three categories of static analysis tools: source code security analyzers, bytecode scanners and binary code scanners. For each of these categories one non-commercial tool is selected for evaluation.

A. *Source Code Security Analyzers*

A source code security analyzer examines source code to detect and report weaknesses that can lead to security vulnerabilities. To use this type of static analysis tools the source code must be available and thus this type of security testing is to be done during the software development lifecycle, or for open-source applications.

OWASP Yasca Project

The OWASP Yasca Project is a source code security analyzer tool, and it stands for Yet Another Source Code Analyzer. It is an open source tool that looks for security vulnerabilities, code-quality, performance, and conformance to best practices in program source code. It leverages external open source tools, such as FindBugs, PMD, JLint, JavaScript Lint, PHPLint, Cppcheck, ClamAV, RATS, and Pixy to scan specific file types, and also contains many custom scanners developed just for Yasca. Yasca is easily extensible and includes a large number of custom rules implemented via a plug-in-based architecture.

B. *Bytecode Scanners*

Tools in this category are used just like tools in source code security analyzers, but instead of detecting the weaknesses in the source code the weaknesses are detected in the bytecode. If the source code is not available, bytecode scanners can be used to find the weaknesses.

FindBugs

FindBugs is an open-source static bytecode analyzer for Java, developed at the University of Maryland. FindBugs uses static analysis to inspect Java bytecode for occurrences of bug patterns. Static analysis means that FindBugs can find bugs by simply inspecting a program's code; executing the program is not necessary. FindBugs supports a plug-in architecture allowing anyone to add new bug detectors.

C. *Binary Code Scanners*

Binary Code Scanners work at the binary code level after the code has been compiled which allows the tool to find vulnerabilities in the compiled and imported binary libraries. Binary code scanners analyze machine code to model a language-neutral representation of the program's behavior, control, and data flow, call trees and external function calls. Then this model traversed by automated vulnerability scanners to find vulnerabilities [26].

Microsoft Code Analysis Tool .NET (CAT.NET)

CAT.NET is a binary code analysis tool from Microsoft. It helps to identify security flaws within a managed code (C#, Visual Basic .NET, J#) application. CAT.NET scans the binary and/or assembly code of the application, and traces the data flow among its statements, methods, and assemblies. This includes indirect data types such as property assignments and instance tainting operations. The engine works by reading the target assembly and all reference assemblies used in the application and then analyzing all of the methods contained within each. It finally displays the issues it finds in a list that you can use to jump directly to the places in your application's source code where those issues were found.

IV. EVALUATION METHODOLOGY

The evaluation methodology used in this research is based on the evaluation methodology defined by NIST SAMATE project for evaluating software assurance tools. The NIST SAMATE project provides a resource for source code security analysis, web application vulnerability scanners and binary code scanners. Also SAMATE provides a Reference Dataset, which is a community repository of example code and other artifacts to help end users evaluate tools and developers test their methods. Resources in NIST SAMATE project focus on C and C++. This research addresses other widely used languages and technologies such as Java and .Net.

The problem is approached by identifying common software vulnerabilities and weaknesses based on CWE/SANS Top 25, OWASP Top Ten and NIST source code weaknesses. After selecting the common weaknesses, general test plans and test suites are defined for each of the three classes of security static analysis tools: source code security analyzers, binary code scanners and bytecode scanners. Experiments are then executed on the selected tools: Yasca, CAT.NET and FindBugs. After running test plans, results are analyzed to evaluate the effectiveness of the tools in covering common software vulnerabilities and weaknesses.

A. *Defining Common Software Vulnerabilities*

There are two types of vulnerabilities in software: bug and flaw. A bug is a vulnerability caused by a problem with the syntax of the code and it appears in implementation phase. A flaw is a vulnerability caused by a non-syntax problem and it appears in design phase. Several organizations are involved in maintaining categories of common software vulnerabilities. MITRE and individual researchers and representatives from organizations in academia, commercial sector and government initiated Common Weakness Enumeration (CWE). CWE is a community-developed formal list of common software weaknesses. CWE provides a unified, measurable set of software weaknesses that is enabling more effective discussion, description, selection, and use of software security tools and services that can find these weaknesses in source code and operational systems as well as better understanding and management of software weaknesses related to architecture and design. Krutz & Fry [26] summarize the benefits offered by CWE in the following points:

- Servers as a common language for describing software security weaknesses in architecture, design or code;
- Serves as a standard measuring stick for software security tools targeting these weaknesses; and
- Provides a common baseline standard for weakness identification, mitigation and prevention efforts.

For the purpose of this study, only nine of common weaknesses are selected. The weaknesses that are not applicable to Java and .Net technologies were eliminated. Also this research does not cover three of the common weaknesses, which are: CWE-306, CWE-601 and CWE-327. Table 2 shows the selected common weaknesses covered in the scope of this research.

Table 2: Selected common software weaknesses

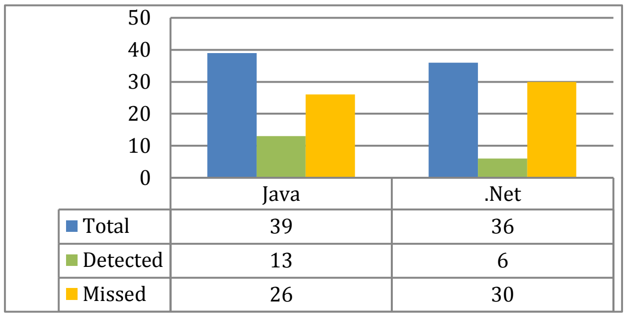
CWE-ID	Name
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-798	Use of Hard-coded Credentials
CWE-362	Race Condition
CWE-352	Cross-Site Request Forgery (CSRF)
CWE-285	Improper Access Control (Authorization)
CWE-311	Missing Encryption of Sensitive Data
CWE-209	Information Exposure Through an Error Message

V. Results and Analysis

This section presents analysis and discussion of the results produced from experiments on the three selected tools for security static analysis. Each of the tools analyzed the same weaknesses in different level: source code, bytecode or binary code.

A. *Analysis of Results from Yasca*

The total number of the test cases used to evaluate Yasca was 75 test cases. 39 of these test cases were for Java test suite and 36 for .Net test suite. As illustrated in Figure 2, Yasca was able to detect weaknesses in 13 Java test cases out of 39 Java test cases. On the other hand Yasca was able to detect 6 weaknesses out of 36 .Net test cases. In terms of the number of weaknesses detected, Yasca was more effective with Java test cases.



	Java	.Net
■ Total	39	36
■ Detected	13	6
■ Missed	26	30

Figure 2: Yasca results summary

From the perspective of the number of weakness categories, Yasca detected four weakness categories in Java test cases and two weakness categories in .Net test cases. In Java test cases Yasca detected following weaknesses:

- CWE-78: OS Command Injection;
- CWE-79: Cross-site Scripting;
- CWE-89: SQL Injection; and
- CWE-209: Information Exposure Through an Error Message.

Figure 3 illustrates number of Java test cases used for each of the above weakness categories, number of detected weaknesses and number of missed weaknesses.

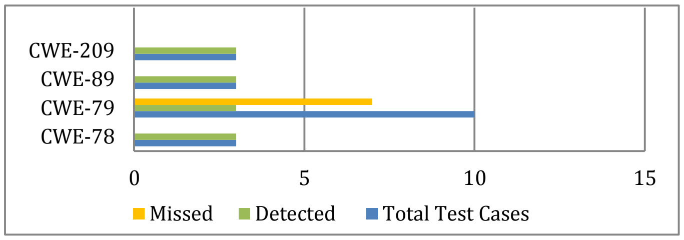


Figure 3: Weaknesses detected in Java test cases

For .Net test cases Yasca detected the following weaknesses:

- CWE-79: Cross-site Scripting; and
- CWE-89: SQL Injection.

Figure 4 illustrates the number of .Net test cases used for each of the above weakness categories, number of detected weaknesses and number of missed weaknesses.

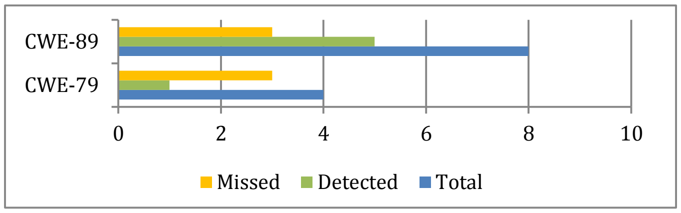


Figure 4: Weaknesses detected in .Net test cases

Yasca showed better results with Java test cases. It was able to detect more weaknesses from four weakness categories in Java test cases. The fact that Yasca is an aggregator of other open source tools is the reason for having less effectiveness in .Net test cases in comparison with Java test cases. The current version of Yasca aggregates more than one source code analyzers for Java like: PMP, JLint and FindBugs. For .Net Yasca currently aggregates only FxCop. The lack of open source tools for .Net security static analysis leads to limited .Net analysis activities in Yasca.

B. *Analysis of Results from FindBugs*

The total number of Java bytecode test cases used to evaluate FindBugs was 34 test cases. The tool was able to detect weaknesses in 7 test cases. The weaknesses detected in FindBugs are from three weakness categories which are:

- CWE-79: Cross-site Scripting;
- CWE-89: SQL Injection; and
- CWE-798: Hard-coded Credentials.

Figure 5 illustrates the number of Java bytecode test cases used for each of the above weaknesses categories, number of detected weaknesses and number of missed weaknesses.

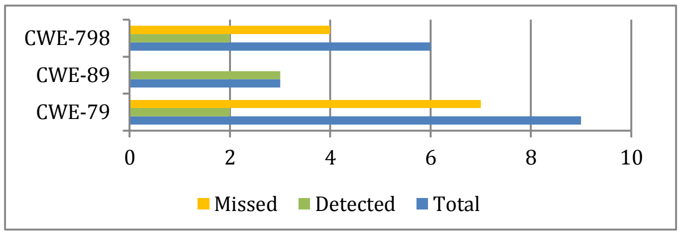


Figure 5: Weaknesses detected in FindBugs

FindBugs detected fewer weaknesses in Java test cases than source code analyzer (Yasca), however FindBugs was able to detect hard-coded credentials weaknesses (CWE-798) that Yasca was not able to detect. FindBugs detected all SQL injection weaknesses. It detected two out of 9 cross-site scripting, one in basic structure and one in loop structure, but it did not detect any cross-site scripting weakness in scope structure. For hard-coded credentials FindBugs detected 2 out of 6 weaknesses, one in basic structure and one in loop structure, but it did not detect any hard-coded credential weakness in scope structure.

C. *Analysis of Results from CAT.NET*

CAT.NET scanned 31 dll and exe files that were generated from .Net source code test cases that implement the nine weaknesses. CAT.NET was not able to detect any of the weaknesses in these files. The test plan for binary code scanner was extended to include more test cases. Six new test cases were developed for cross-site scripting and SQL injection. CAT.NET was able to detect all the weaknesses in binary files from the extended test plan.

VI. Conclusion and Future Work

This paper introduced an evaluation framework for security static analysis tools. The framework provides test plans for evaluating source code analyzers, bytecode scanners and binary code scanners. Each test plan includes set of test cases that implement security weaknesses. Test plans for source code analyzers covered both Java and .Net test cases. Bytecode scanners test plan covered only Java test cases, and binary code test plan covered .Net test cases. For future work, we plan to extend the test plans to include more test suites to calculate the false positive ratio generated by the tools. Test plans can be extended also to cover more software weaknesses and to include test suites for mobile applications vulnerabilities [30]. The evaluation can be extended to cover commercial and dynamic analysis tools. Further research can be done on how to enhance the open source tools like Yasca by writing new security rules to cover more security weaknesses in source code and develop Yasca plug-ins for open source IDEs like NetBeans and Eclipse.

References

[1] Talukder, A. K., Maurya, V. K., Santhosh, B. G., Ebenezer, J., Jevitha, K. P., Samanta, S., et al. (2009). Security-aware software development lifecycle (SaSDLC): processes and tools. Proceedings of the Sixth international conference on Wireless and Optical Communications Networks (pp. 253-257). Cairo, Egypt: IEEE Press.

[2] Peterson, G. (2007). *Security Architecture Blueprint.* Arctec Group.

[3] McGraw, G., & Viega, J. (2001). *Building Secure Software.* Addison Wesley.

[4] Wang, J. A., Wang, H., Guo, M., & Xia, M. (2009). Security Metrics for Software System. Proceedings of the 47th Annual Southeast Regional Conference. New York: ACM.

[5] Jones, R. L., & Rastogi, A. (2004). Secure Coding: Building Security into the Software Development Lifecycle. *Information Security Journal: A Global Perspective* , 29-39.

[6] Gilliam, D. P., Wolfe, T. L., Sherif, J. S., & Bishop, M. (2003). Software Security Checklist for the Software Lifecycle. *Proceedings of the 12th IInternational Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprise* (pp. 243–248).

[7] McGraw, G. (Gust). (2008). How to Start a Secure Software Development Program. CERT's Podcast Series. Podcast retrieved from http://www.cert.org/podcast/mp3/2/20080820mcgraw-full.mp3.

[8] Fogie, S., & McGraw, G. (2004). *The U.S. Government, and Good vs. Evil.* Retrieved Jan 15, 2013, from InformIT: http://www.informit.com/articles/article.aspx?p=174303.

[9] Howard, M., & Lipner, S. (2006). *The Security Development Lifecycle.* Microsoft Press.

[10] Lipner, S. B. (2005). Building More Secure Commercial Software. *GI-Jahrestagung* , 21-28.

[11] Gates, B. (2002). *Trustworthy Computing.* Retrieved Jan 11, 2013, from Microsoft Executive E-mail: http://www.microsoft.com/mscorp/execmail/2002/07-18twc.mspx.

[12] Zhang, J., Zhang, L.-J., & Chung, J.-Y. (2004). WS-Trustworthy: A Framework for Web Services Centered Trustworthy Computing. Proceedings 2004 IEEE International Conference on Services Computing, (pp. 186 - 193).

[13] Fiebig, C. (2003). *The Journey to Trustworthy Computing.*. Retrieved Jan 16, 2013, from Microsoft News Center: http://www.microsoft.com/presspass/features/2003/jan03/01-15twcanniversary.mspx.

[14] McGraw, G. (2004). Software Security. *IEEE Security & Privacy*.

[15] Ruefle, R. (2007). The Role of Computer Security Incident Response Teams in the Software Development Lifecycle. Retrieved Jan 16, 2013, from Build Security In: https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/incident/661-BSI.html.

[16] Lipner, S. (2004). The trustworthy computing security development lifecycle. *Proceedings of the 20th Annual Computer Security Applications Conference* (pp. 2-13). IEEE Computer Society.

[17] Mead, N.R.; Hilburn, T.B., "Building Security In: Preparing for a Software Security Career," *Security & Privacy, IEEE* , vol.11, no.6, pp.80,83, Nov.-Dec. 2013

[18] Misra, A. (2013). *Core Software Security*. Auerbach Publications.

[19] Tondel, I. A., Jaatun, M. G., & Meland, P. H. (2008). Security Requirements for the Rest of Us: A Survey. IEEE Software , 20-27.

[20] Grance, T., Hash, J., & Stevens, M. (2004). Security considerations in the information system development life cycle. NIST.

[21] Microsoft. (2011). History of the SDL. Retrieved Jan 11, 2013, from Microsoft Security Development Lifecycle: http://www.microsoft.com/security/sdl/about/history.aspx.Microsoft. (2012). SDL Process Guidance Version 5.2. Microsoft.

[22] Microsoft. (2009). Security Development Lifecycle for Agile Development. Microsoft.

[23] OWASP CLASP Project. (2009). Retrieved Jan 19, 2013, from OWASP: http://www.owasp.org/index.php/Category:OWASP_CLASP_Project.

[24] Gregoire, J., Buyens, K., Win, B. D., Scandariato, R., & Joose, W. (2007). On the Secure Software Development Process: CLASP and SDL Compared. *Third International Workshop*.

[25] Krutz, R. L., & Fry, A. J. (2009). The CSSLP Prep Guide: Matering the Certified Secure Software Lifecycle Professional. Indiana: Wiley.

[26] McGraw, G., & Chess, B. (2008). Working Towards a Realistic Maturity Model. Retrieved Oct 13, 2014, from InformIT: http://www.informit.com/articles/article.aspx?p=1271382.

[27] Davis, N. (2006). *Secure Software Development Lifecycle Processes.* Retrieved Jan 12, 2013, from Build Security IN: https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/sdlc/326-BSI.html.

[28] NIST. (2005). Software Assurance Metrics And Tool Evaluation. Retrieved Jan 12, 2013, from Software Assurance Metrics And Tool Evaluation: http://samate.nist.gov/Main_Page.html.

[29] Hossain Shahriar and Mohammad Zulkernine. "Mitigating program security vulnerabilities: Approaches and challenges", ACM Computing Surveys (CSUR), Volume 44 Issue 3, June 2012, Article No. 11

[30] OWASP Mobile Security.(2014). Retrieved Oct 13, 2014, from OWASP: https://www.owasp.org/index.php/OWASP_Mobile_Security_Project.