NIST Special Publication 500-268 v1.1

Source Code Security Analysis Tool Functional Specification Version 1.1

Paul E. Black
Michael Kass
Michael Koo
Elizabeth Fong

Software and Systems Division
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

February 2011

U.S. Department of Commerce
Gary Locke, *Secretary*

National Institute of Standards and Technology
Dr. Patrick D. Gallagher, Director

Abstract:

Software assurance tools are a fundamental resource to improve quality in today's software applications. Some tools analyze software requirements or design models to help determine if an application is secure. Others analyze source code or executables. This document specifies the behavior of one class of software assurance tool: the source code security analyzer. Because many software security weaknesses are introduced at the implementation phase, using a source code security analyzer should help reduce the number of security vulnerabilities in software. This specification defines a minimum capability to help software professionals understand how a tool can help meet their software security assurance needs.

Keywords:

Homeland security; software assurance tools; source code analysis; vulnerability.

Changes to this version:

This version 1.1 updates version 1.0 by adding the SPARK language in Annex A and improving explanations.

Table of Contents

1.0 Introduction

1.1 Purpose

The National Institute of Standards and Technology (NIST) is working with the U.S. Department of Homeland Security's National Cybersecurity Division to improve the state of the practice in software assurance. Through the development of tool functional specifications, test suites and tool metrics, the NIST Software Assurance Metrics and Tool Evaluation, or SAMATE, project aims to better characterize the state of the art for different classes of software security assurance tools.

Source code security analysis tools scan a textual (human readable) version of source files that comprise a portion or all of an application program. These files may contain inadvertent or deliberate weaknesses that could lead to security vulnerabilities in the executable versions of the application program. This document specifies a set of functional feature requirements for a source code security analysis tool or set of tools, including a list of common code weaknesses that account for many of today's vulnerabilities.

This specification, together with the corresponding test plan and test suite, serves as a guide to understanding the capability of source code security analysis tools against this set of weaknesses. Many useful tools do not attempt to identify all of the weaknesses listed in this specification. The goal of this specification is not to prescribe the features and functions that all source code security analysis tools must have. The goal is to identify code weaknesses that significantly affect the security of software applications today and provide a user of such tools with a way to determine if, and how well a tool, or combination of tools, identifies these particular weaknesses.

Use of a tool or toolkit that complies with this specification does not guarantee the code will be free of weaknesses. It does however provide a tool user with knowledge that their tool solution covers some of the most prevalent and highly exploitable security weaknesses.

1.2 Scope

This specification is limited to software tools that examine source code files for security weaknesses and potential vulnerabilities. Tools that scan other artifacts, like requirements, bytecode or binary code, and tools that dynamically execute code are outside the scope. Annex A of this document, Source Code Weaknesses, specifically addresses C, C++, Java, and SPARK [Barnes] source code.

We started with C, C++, and Java because they are the languages in which most of today's vulnerabilities have been identified and on which most source code security analysis tools focus. These weaknesses may exist in other languages as well.

There are languages that are, by design, more suitable for secure programming. We added SPARK as an example of one. Such languages entirely preclude many common weaknesses and minimize or expose others. Choosing such languages mitigates many security risks.

This document specifies core functionality only. Critical production tools should have capabilities far beyond those indicated here. Many important attributes, like compatibility with integrated development environments or IDEs and ease of use, are not addressed.

The misuse or proper use of a tool is outside the scope of this specification.

The issues and challenges in engineering secure systems and their software are outside the scope of this specification.

1.3 Audience

The target audiences for this specification are users and evaluators of source code security analysis tools. It may also be useful to software assurance researchers, and developers of source code security analysis tools.

1.4 Technical Background

This section gives some technical background, defines terms we use in this specification, explains how concepts designated by those terms are related, and details some challenges in source code analysis for security assurance.

The Role of Source Code Analysis in Software Assurance

No amount of analysis and patching can imbue software with high levels of security, quality, correctness, or other important properties. Such properties must be designed in and built in. Good choices of language, platform, and discipline are worth orders of magnitude more than reactive efforts. Nevertheless testing or examination of code has benefits.

For instance, to determine how different methods or processes affect the quality of the resultant code, the code can be examined. If the origin of code has limited visibility, testing or static analysis are the only ways to gain higher assurance. Existing, legacy code must be examined to assess its quality and determine what, if any, remediation is needed.

Testing, or dynamic analysis, has the advantage of examining the behavior of software in operation. In contrast, only static analysis can be expected to find malicious trapdoors. Analysis of binary or executable code, including bytecode, avoids assumptions about compilation or source code semantics. Only the binary may be available for libraries or purchased software. However, source code security analysis can give developers feedback on better practices.

Remediation is often done in source code. Analysis of higher-level constructs, such as models, designs, use cases, or requirements documents, is possible, too. However, these higher-level artifacts often lack rigor and rarely reflect all the critical detail in source code implementations. Thus static analysis of source code is a reasonable place to work for higher software assurance.

Terms Used in This Specification

Often, different terms are used to refer to the same concept in the software assurance and security literature. Different authors may use the same term to refer to different concepts. For the purposes of this document, the following terms and definitions apply. To begin, any event that is a violation of a particular system's explicit (or implicit) security policy is a *security failure*, or simply, failure. For example, if an unauthorized person gains "root" or "admin" privileges, security has failed. Similarly, if unauthorized people can read Social Security numbers from your web site, security has failed.

A *vulnerability* is a property of system security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a security failure. (After [SP800-27]) In our model, the source of any failure is a latent vulnerability. In other words, if there is a failure, there must have been a vulnerability. A vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation.

In the unauthorized privileges example above, the combination of the two weaknesses of allowing weak passwords and of not locking out an account after repeated password mismatches constitute the vulnerability. This vulnerability can be exploited by a brute force attack to cause the failure of an unauthorized person gaining elevated privileges. An SQL injection vulnerability might be exploited several different ways to produce different failures, such as dropping a table or revealing all its contents. If spyware can steal a user's password, it is a vulnerability. But it may be hard to attribute the vulnerability to a few lines of code that can be "fixed." Spyware typically exploits system weaknesses, which require changes at the system level.

Sometimes a weakness can never result in a failure, in which case it is not exploitable and not a vulnerability. Such a weakness might be masked by another part of the software or might only cause a failure in combination with another weakness. Thus we use the term "weakness" instead of "flaw" or "defect."

For several reasons no tool can correctly determine in every conceivable case whether or not a piece of code has a vulnerability. First, a weakness may result in a vulnerability in one environment, but not in another. Second, Rice proved [Rice] that no algorithm can correctly decide in every case whether or not a

piece of code has a property, such as a weakness. Third, practical analysis algorithms have limits because of performance, approximations, and intellectual investment. Some vulnerabilities can only be identified if a tool performs inter-file, inter-procedural, or flow-sensitive analysis of the code. Each different *code complexity*, such as fixed or variable loops, memory indexing nested within indexing, local vs. global scope, and others listed in Annex B, may require additional analytical capabilities. Deliberate obfuscation with convoluted code structures makes the analysis even harder. Fourth, a tool may not have "rules" to find all known vulnerabilities. Worse, new exploits are being invented and new vulnerabilities recognized all the time.

Since no tool can be omniscient, a tool may be written to be cautious and report questionable constructs. Some of those reports may turn out to be false alarms or *false positives*. To reduce wasting users' time on false alarms, a tool may be written to only report constructs that are (almost) certainly vulnerabilities. In this case it may miss some vulnerabilities. A missed vulnerability is called a *false negative*. A tool may do a more detailed or precise analysis, which is computationally intensive, to reduce both false alarms and missed vulnerabilities. The ideal is a tool that reports all real vulnerabilities (no false negatives) with no false alarms. Although this is impossible even in theory, tools may use a combination of approaches to balance performance, false alarms, and missed vulnerabilities. Since a failure only takes one vulnerability, the requirements have a tone of catching all weaknesses. Practical considerations require the *false positive rate* [Fleiss] to be acceptably low for the domain.

A tool may grade weaknesses according to severity, potential for exploit, certainty that they are vulnerabilities, etc. Ultimately people must analyze the tool's report and the code then decide

- which reported items are not true vulnerabilities,
- which items are acceptable risks and will not be mitigated, and
- which items to mitigate, and how to mitigate them.

To save analysis time in later runs, some tools allow the user to *suppress* weakness instances so they are not reported again.

1.5 Glossary of Terms

This glossary provides descriptions for terms used in this document.

Name	Description
false negative	When a tool does not report a weakness where one is present. If the tool does not claim to identify a certain class of weakness, not reporting a weakness of that class is not a false negative.
false positive	When a tool reports a weakness where no weakness is present.
false positive rate	The number of false positives divided by the sum of the number of false positives and the number of true positives.
flow-sensitive analysis	Analysis of a computer program that takes into account the flow of control.
inter-file analysis	Analysis of code residing in different files that have procedural, data, or other interdependencies.
inter-procedural analysis	Analysis between calling and called procedures within a computer program.
security failure	Any event that is a violation of a particular system's explicit or implicit security policy.
security vulnerability	A property of system requirements, design, implementation, or operation that could be accidentally triggered or intentionally

	exploited and result in a security failure.
source code	A series of statements written in a human-readable computer programming language.
true positive	When a tool reports a weakness where one is present.
weakness	A piece of code that may lead to a vulnerability.
weakness suppression system	A feature that permits the user to flag a line of code not to be reported by the tool in subsequent scans.

2.0 Functional Requirements

2.1 High Level View

Informally, what does a source code security analysis tool or tool set do? At a minimum the tool(s) should:

- Identify a select set of classes of software security weaknesses in source code.
- Report the security weaknesses that it identifies, what kind of weakness each one is, and where each one is located.
- Not report too many false positives.

Optionally a tool should:

- Produce a report compatible with other tools, for instance in the Software Assurance Findings Expression Schema (SAFES) format [Barnum].
- Allow the user to suppress reporting of selected weaknesses.
- Use standard names for weakness classes.

2.2 Requirements for Mandatory Features

To meet a core capability, a source code security analysis tool or set of tools must be able to accomplish the tasks described below. The tool(s) shall:

SCSA-RM-1: Identify all of the classes of weaknesses listed in Annex A.

SCSA-RM-2: Textually report any weaknesses that it identifies.

SCSA-RM-3: For any identified weaknesses in the classes listed in Annex A, report the class using a semantically equivalent name.

SCSA-RM-4: For any identified weaknesses, report at least one location by providing the directory path, file name and line number.

SCSA-RM-5: Identify weaknesses despite the presence of the coding complexities listed in Annex B.

SCSA-RM-6: Have an acceptably low false positive rate.

2.3 Requirements for Optional Features

The following requirements apply to optional tool features. If the tool supports an optional feature, then the requirement for that feature applies, and the tool can be tested against it. A specific tool might optionally provide none, some, or all of the features described by these requirements. Optionally, the tool(s) shall:

SCSA-RO-1: Produce an XML-formatted report.

SCSA-RO-2: Not report a weakness instance that has been suppressed.

SCSA-RO-3: Use the Common Weakness Enumeration [CWE] number and name of the weakness class it reports.

3.0 References

[Barnes] Barnes, John, *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.

[Barnum] Barnum, Sean, *Taming the Tower of Babel: Software Assurance Findings Expression Schema (SAFES) Framework*, 12th Semi-Annual Software Assurance Forum, March 2010, https://buildsecurityin.us-cert.gov/swa/forum_march_2010.html

[CVE] Common Vulnerability and Exposures, the MITRE Corporation, http://cve.mitre.org/

[CWE] Common Weakness Enumeration, the MITRE Corporation, http://cwe.mitre.org/

[Fleiss] Fleiss, Joseph L. (1981). *Statistical Methods for Rates and Proportions*, 2nd ed., John Wiley and Sons, New York, pp 4-8.

[Kratkiewicz] Kratkiewicz, Kendra. (2005). *Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code*, Master's Thesis, Harvard University, Cambridge, MA, 285 pages. http://www.ll.mit.edu/IST/pubs/KratkiewiczThesis.pdf

[Rice] Rice, Henry Gordon, *Classes of Recursively Enumerable Sets and Their Decision Problems*, Transactions of the American Mathematical Society, 74(2):358-366, March 1953.

[SP800-27] Engineering Principles for Information Technology Security (A Baseline for Achieving Security), NIST SP 800-27, Revision A, June 2004. http://csrc.nist.gov/publications/nistpubs/

[Tokar] Tokar, Joyce L, Jones, F. David, Black, Paul E., and Duplika, Chris E., *Software Vulnerabilities Precluded by SPARK*, (to be published)Paul.

[XML] Extensible Markup Language, World Wide Web Consortium (W3C), http://www.w3.org/XML/

Annex A Source Code Weaknesses

The classes of source code weaknesses listed in this table represent a “base set” of code weaknesses. Criteria for selection of weaknesses include:

- **Found in existing code today** – Corresponding vulnerabilities are found in existing software applications.
- **Recognized by tools today -** Tools today are able to identify these weaknesses in source code and identify their associated file names and line numbers.
- **Likelihood of exploit is medium to high –** The vulnerability is fairly easy for a malicious user to recognize and to exploit.

For each weakness, this table contains the CWE name and identifier number, a short description, the relevant language(s) in which it might occur, and the code complexities that apply. For other relevant information, such as likelihood of exploitation, instances in the common vulnerability and exposures list [CVE], background, consequences, and remediation; please see [CWE]. For SPARK, some weaknesses cannot occur depending upon the toolset used; please see [Barnes, Tokar].

Name	CWE ID	Description	Language(s)	Relevant Complexities
Input Validation				
Basic XSS	80	Inadequately filtered input, allows a malicious script to be passed to a web application that in turn passes it to another client.	C,C++, Java, SPARK	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Resource Injection	99	Inadequately filtered input is used in an argument to a resource operation function.	C, C++, Java, SPARK	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
OS Command Injection	78	Inadequately filtered input is used in an argument to a system operation execution function.	C, C++, Java, SPARK	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
SQL Injection	89	Inadequately filtered input is used in an argument to a SQL command calling function.	C, C++, Java, SPARK	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Range Errors				
Stack overflow	121	Input is used in an argument to create or copy data beyond the fixed memory boundary of a buffer on the stack.	C, C++	All
Heap overflow	122	Input is used in an argument to create or copy beyond the fixed memory boundary of a buffer in the heap portion of memory.	C, C++	All

Format string vulnerability	134	Inadequately filtered input is used to format data in printf() style C/C++ functions.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Improper Null Termination	170	The software does not properly terminate a string.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
API Abuse				
Heap Inspection	244	Using realloc() to resize buffers that store sensitive information can leave the information exposed because it is not removed.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Often Misused: String Management	251	Some string manipulation functions can be exploited through their input to produce buffer overflows.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Security Features				
Hard-Coded Password	259	Hard-coded data is used to authenticate or passed to a login function.	C/C++, Java, SPARK	scope, address alias level, container, local control flow, loop structure, buffer address type
Time and State				
Time-of-check Time-of-use race condition	367	Between the time that a resource (or its reference) is checked and the time it is used, a change may occur in the resource to invalidate the check.	C, C++, Java, *SPARK**	asynchronous
Unchecked Error Condition	391	No action is taken after an error or exception occurs.	C, C++, Java	none
Code Quality				
Memory leak	401	Memory is allocated, but is not released after its final used.	C, C++	scope, address alias level, container, local control flow, loop structure
Unrestricted Critical Resource Lock	412	A resource may locked by an unauthorized external agent.	C, C++, Java, *SPARK**	asynchronous
Double Free	415	An attempt is made to free memory that has previously been used in a free() function call.	C, C++	scope, address alias level, container, local control flow, loop structure, buffer address type
Use After Free	416	An attempt is made to access memory previously released by a call to the	C, C++	scope, address alias level, container, local control flow, loop structure, buffer address type

		free() function.		
Uninitialized variable	457	A variable is created without assigning it a value and is subsequently referenced in the program.	C, C++	scope, address alias level, container, local control flow, loop structure
Unintentional pointer scaling	468	Improper mixing of pointer types in an expression may result in references to memory beyond that intended by the program.	C, C++	data type
Null Dereference	476	A pointer with a value of NULL is used as though it pointed to a valid memory area.	C, C++	taint, scope, address alias level, container, local control flow, loop structure
Encapsulation				
Leftover Debug Code	489	Debug code can create unintended entry points in an application.	C, C++, Java, SPARK	none

* This weakness can only occur in certain cases if RavenSPARK is used.

Annex B Code Complexity Variations

To locate and identify source code weaknesses listed in Annex A, a source code security analysis tool must be able to find those weaknesses within relevant complex coding structures. A list of these types of structures, adapted from [Kratkiewicz], is provided below. Some of the complexities are language specific (e.g. the use of pointers in C, C++), however, most exist in C, C++ and Java. Equivalent constructs in other languages will be added, as tools for those languages are addressed in this specification.

Complexity	Description	Enumeration
address alias level	level of “indirection” of buffer alias using variable(s) containing the address	1, 2, etc.
array address complexity	level of complexity of the address value of an array buffer	constant, variable, linear expression, nonlinear expression, function return value, array content value
array index complexity	level of complexity of the index value of an array buffer using variable assignment	constant, variable, linear expression, nonlinear expression, function return value, array content value
array length/limit complexity	level of complexity of the index of an array buffer’s length or limit value	constant, variable, linear expression, nonlinear expression, function return value, array content value
asynchronous	asynchronous coding construct	threads, forked process, signal handler
buffer address type	method used to address buffer	pointer, array index
Container	containing data structure	array, struct, union, array of structs, array of unions
data type	type of data read or written	character, integer, floating point, wide character, pointer, unsigned character, unsigned integer
index alias level	level of buffer index alias indirection	1, 2, etc.
local control flow	type of control flow around weakness	if, switch/case, cond (?:), goto/label, setjmp, longjmp, function pointer, recursion
loop complexity	component of loop that is complex	initialization, test, increment
loop iteration	type of loop iteration/termination	fixed, indefinite
loop structure	type of loop construct in which weakness is embedded	standard for, standard do while, standard while, non standard for, non standard do while, non standard while
memory access	type of memory access related to weakness	read, write
memory location	type of memory location related to weakness	heap, stack, data region, BSS, shared memory

Scope	scope of control flow related to weakness	local, within-file/inter-procedural, within-file/global, inter-file/inter-procedural, inter-file/global, inter-class
Taint	type of tainting to input data	argc/argv, environment variables, file or stdin, socket, process environment