DRAFT Source Code Analysis Tool Functional Specification

Information Technology Laboratory (ITL), Software Diagnostics and Conformance Testing Division

15 September, 2006

National Institute of Standards and Technology
Information Technology Laboratory
Software Diagnostics and Conformance Testing Division

Abstract:

Software assurance tools are a fundamental resource for providing an assurance argument for today's software applications throughout the software development lifecycle (SDLC). Software requirements, design models, source code and executable code are analyzed by tools to determine if an application is truly secure. This document specifies the functional behavior of one class of software assurance tool: the source code analyzer. Because the majority of software weaknesses today are introduced at the implementation phase, a specification that defines a "baseline" source code analysis tool capability can help software professionals select a tool that will meet their software assurance needs.

Errata to this version:

None

National Institute of Standards and Technology

Information Technology Laboratory

Software Diagnostics and Conformance Testing Division

Table of Contents

National Institute of Standards and Technology
Information Technology Laboratory
Software Diagnostics and Conformance Testing Division

1 Introduction

The National Institute of Standards and Technology (NIST) is working with the U.S. Department of Homeland Security to determine what the state of the art (SOA) is in software assurance (SwA) tools today. Through the development of tool functional specifications, test suites and tool metrics, the NIST Software Assurance Metrics and Tool Evaluation (SAMATE) project aims to better quantify the state of the art for all classes of software assurance tools.

1.1 Purpose

This document specifies basic, fundamental functional requirements for source code analysis tools used in software assurance evaluations. Production tools should have capabilities far beyond those indicated here. Many important attributes, like cost and ease of use, are not covered.

Accompanying documents detail test cases and methods to ascertain to what extent a tool meets these requirements. The functionality described herein may be embedded in a larger tool with more functionality. The functionality could also be covered by several specialized tools.

Source code analysis tools scan a textual (human readable) version of files that comprise a portion or all of an application program (i.e. the files written in a particular programming language). These files may contain inadvertent or deliberate programming weaknesses that could lead to security vulnerability in the executable version of the application program. Source code analysis tools provide one line of defense against such scenarios. Generally, source code analysis tools are used in combination with good software development practices and other software assurance tools to provide a stronger argument for the security of an application.

The functional requirements are the basis for developing test cases to measure the effectiveness of source code analysis tools. Each functional requirement will have one or more corresponding test cases in the SAMATE Reference Dataset (SRD), detailed procedures for executing the test, and expected test results.

1.2 Scope

This specification is limited to general-purpose production 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 execute code are outside the scope. Appendix A of this document, Source Code Weaknesses, specifically addresses C, C++, and Java source code, although it is recognized that particular weaknesses may exist in other languages as well.

This document specifies baseline functionality. Critical production tools should have capabilities far beyond those indicated here. Many important attributes, like compatibility with integrated development environments (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 audience for this specification is:

- Source code analysis and software assurance researchers
- Implementers and developers of source code analysis tools
- Users and evaluators of source code 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.

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

Code must be analyzed to determine how different methods or processes affect the quality of the resultant code. 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 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.

Often, different terms are used to refer to the same concept in software assurance and security literature. Different authors may use the same term to refer to different concepts. For clarity we give our definitions. To begin any event which 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 or if Social Security numbers can be read through the World Wide Web by unauthorized people, 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 [NIST SP 800-27]) In our model the source of any failure is a latent vulnerability. 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 allow 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 particular weaknesses in software that can be "fixed." Spyware typically exploits system weaknesses, which require changes at the system level.

Sometimes a weakness cannot result in a failure, in which case it is not exploitable and not a vulnerability. Such a weakness may be masked by another part of the software or it may only cause a failure in combination with another weakness. Thus we use the term "weakness" instead of "flaw" or "defect."

A source code analysis tool examines software and reports weaknesses it finds. They may be graded according to severity, potential for exploit, certainty that they result in vulnerabilities, etc. Ultimately people must use the reports to 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.

The report may even lead the user to reject a piece of software altogether as insufficiently secure to use or as needing to be discarded and written from scratch.

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 that no algorithm can correctly decide whether or not a piece of code has a property, such as a weakness, in every case. Third, practical analysis algorithms have limits because of performance and intellectual investment. Some vulnerabilities can only be identified if a tool performs inter-file, inter-procedural, or flow-sensitive analysis of the code. Deliberate obfuscation with complex code structures makes the analysis even harder. Fourth, a tool may not have "rules" to find all known vulnerabilities. This is even harder since new exploits and vulnerabilities are being invented all the time.

Since no tool can be perfect, a tool may be biased on the side of caution and report questionable constructs. Some of those may turn out to be false alarms or *false positives*. To reduce time wasted on false alarms, a tool may be biased on the side of certainty and only report constructs which are (almost) certainly vulnerabilities. In this case it may miss some vulnerabilities. A missed vulnerability is called a *false negative*. Changing the threshold of certainty to report a construct as a vulnerability trades fewer false negatives for more false alarms and vice versa. The ideal would be a tool that reports every real vulnerability (no false negatives) with no false alarms. Even though this is theoretically impossible, utility requires some metric for the tradeoff between false alarms and false negatives.

1.5 Glossary of Terms

This glossary was added to provide context for terms used in this document.

Name	Description
baseline functionality	The minimally acceptable set of functions that a tool shall successfully perform to be considered conformant with this specification.
flow-sensitive analysis	Analysis of changes in logical program flow while maintaining

	information about what facts may or will not hold during the execution of a program.
dynamic analysis	Analysis of a computer program through execution.
false negative	Failure of a tool to report a weakness, when in fact there is one present in the code.
false positive	Reporting of a weakness by a tool, where there is none.
weakness suppression system	A feature of source code analysis tools that permits flagging of a line of code as “ignorable” by the tool in subsequent source code scans.
inter-file analysis	Analysis of code residing in different files.
inter-procedural analysis	Analysis between calling and called procedures within a computer program.
security vulnerability	A property of system security 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.
static program analysis	Analysis of a computer program performed without actually executing the program.
weakness	A defect in a a system that may (or may not) lead to a vulnerability.

2 Functional Requirements

In this section we first give a high-level description of the functional requirements for source code analysis tools, then detail the requirements for mandatory and optional features.

2.1 High Level View

A baseline level of functionality is required in order for a source code analysis tool to be considered conformant with this specification. A source code analysis tool shall be able to (at a minimum):

- Identify a select set of software security weaknesses in source code.
- Generate a text report of the security weaknesses that it finds, indicating the source file name and line number(s) where those weaknesses are located.

2.2 Requirements for Mandatory Features

In order to meet this baseline capability, a source code analysis tool must be able to accomplish the tasks described in the mandatory requirements listed below. The following functional requirements are mandatory and shall be met by all source code analysis tools for the code weaknesses that they claim to identify.

SCA-RM-1: The tool shall identify all of the code weaknesses listed in Appendix A.

SCA-RM-2: The tool shall generate a text report identifying the code weaknesses that it finds.

SCA-RM-3: The tool shall identify the weakness with a name semantically equivalent to those in Appendix A.

SCA-RM-4: The tool shall specify the location of a weakness by providing the directory path, file name and line number.

SCA-RM-5: The tool shall be capable of identifying any weaknesses within all of relevant the coding constructs listed in Appendix B.

SCA-RM-6: The tool shall have an acceptably low “false-positive” ratio.

2.3 Requirements for Optional Features

The following requirements apply to optional tool features. If the tool supports the applicable optional feature, then the requirement for that feature applies, and the tool can be tested against it. This means that a specific tool might optionally provide none, some or all of the features described by these requirements.

SCA-RO-1: The tool shall produce an XML-formatted report.

SCA-RO-2: The tool will not report a weakness instance, which has been suppressed.

SCA-RO-3: If a widely accepted dictionary of weaknesses exists, the tool shall identify a weakness with that dictionary's name and description.

3 References

[CWE] Common Weakness Enumeration, The MITRE Corporation, web site http://cve.mitre.org/cwe/index.html#tree

[MIT] Kratkiewicz, K. (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

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

Appendix A Source Code Weaknesses

The source code weaknesses listed in this table represent a “base set” of code weaknesses that a source code analysis tool (or combination of source code analysis tools) must be able to identify if they support the analysis of the language in which the weakness exists. Criteria for selection of weaknesses include:

- **Found in existing code today** – The weaknesses listed below are found in real 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 weakness is fairly easy for a malicious user to recognize and to exploit.

Because the body of known software weaknesses is evolving (with new ones discovered every day), this list will grow. Additionally, as source code analysis tools mature in their capabilities and are able to identify more software weaknesses, those weaknesses will be added to this list. The names and descriptions in this list are found in [CWE].

Name	Description	Language(s)	Relevant Complexities
Input Validation			
Path Manipulation	Allowing user input to control paths used by the application may enable an attacker to access otherwise protected files.	C, C++, Java	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Cross Site Scripting.Basic XSS	'Basic' XSS involves a complete lack of cleansing of any special characters, including the most fundamental XSS elements such as "<", ">", and "&".	C,C++, Java	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Resource Injection	Allowing user input to control resource identifiers might enable an attacker to access or modify otherwise protected system resources.	C, C++, Java	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
OS Command Injection	Command injection problems are a subset of injection problem, in which the process is tricked into calling external processes of the attackers choice through the injection of control-plane data into the data plane. Also called “shell injection”.	C, C++, Java	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
SQL Injection	SQL injection attacks are another instantiation of injection attack, in	C, C++, Java	taint, scope, address alias level, container, local

	which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.		control flow, loop structure, buffer address type
Range Errors			
Stack overflow	A stack overflow condition is a buffer overflow condition, where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).	C, C++	All
Heap overflow	A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as the POSIX malloc() call.	C, C++	All
Format string vulnerability	Format string problems occur when a user has the ability to control or write completely the format string used to format data in the printf style family of C/C++ functions.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Improper Null Termination	The product does not properly terminate a string or array with a null character or equivalent terminator. Null termination errors frequently occur in two different ways. An off-by-one error could cause a null to be written out of bounds, leading to an overflow. Or, a program could use a strncpy() function call incorrectly, which prevents a null terminator from being added at all. Other scenarios are possible.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
API Abuse			
Heap Inspection	Using realloc() to resize buffers that store sensitive information can leave the sensitive information exposed to attack because it is not removed from memory.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Often Misused: String Management	Functions that manipulate strings encourage buffer overflows.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type

Security Features			
Hard-Coded Password	Storing a password in plaintext may result in a system compromise.	C/C++, Java	scope, address alias level, container, local control flow, loop structure, buffer address type
Time and State			
Time-of-check Time-of-use race condition	Time-of-check, time-of-use race conditions occur when between the time in which a given resource (or its reference) is checked, and the time that resource is used, a change occurs in the resource to invalidate the results of the check.	C, C++, Java	asynchronous
Unchecked Error Condition	Ignoring exceptions and other error conditions may allow an attacker to induce unexpected behavior unnoticed.	C, C++, Java	none
Code Quality			
Memory leak	Most memory leaks result in general software reliability problems, but if an attacker can intentionally trigger a memory leak, the attacker might be able to launch a denial of service attack (by crashing the program) or take advantage of other unexpected program behavior resulting from a low memory condition .	C, C++	scope, address alias level, container, local control flow, loop structure
Unrestricted Critical Resource Lock	A critical resource can be locked or controlled by an attacker, indefinitely, in a way that prevents access to that resource by others, e.g. by obtaining an exclusive lock or mutex, or modifying the permissions of a shared resource. Inconsistent locking discipline can lead to deadlock.	C, C++, Java	asynchronous
Double Free	Calling free() twice on the same value can lead to a buffer overflow.	C, C++	scope, address alias level, container, local control flow, loop structure, buffer address type
Use After Free	Use after free errors sometimes have no effect and other times cause a program to crash.	C, C++	scope, address alias level, container, local control flow, loop structure, buffer address type
Uninitialized variable	Most uninitialized variable issues result in general software reliability problems, but if attackers can	C, C++	scope, address alias level, container, local control flow, loop structure

	intentionally trigger the use of an uninitialized variable, they might be able to launch a denial of service attack by crashing the program.		
Unintentional pointer scaling	In C and C++, one may often accidentally refer to the wrong memory due to the semantics of when math operations are implicitly scaled.	C, C++	data type
Improper pointer subtraction	The subtraction of one pointer from another in order to determine size is dependant on the assumption that both pointers exist in the same memory chunk.	C, C++	scope, address alias level, container, local control flow, loop structure
Null Dereference	Using the NULL value of a dereferenced pointer as though it were a valid memory address	C, C++	taint, scope, address alias level, container, local control flow, loop structure
Encapsulation			
Private Array-Typed Field Returned From A Public Method	The contents of a private array may be altered unexpectedly through a reference returned from a public method.	Java, C++	scope, address alias level, container, local control flow, loop structure
Public Data Assigned to Private Array-Typed Field	Assigning public data to a private array is equivalent giving public access to the array.	Java, C++	scope, address alias level, container, local control flow, loop structure
Overflow of static internal buffer	A non-final static field can be viewed and edited in dangerous ways.	Java, C++	scope, address alias level, container, local control flow, loop structure
Leftover Debug Code	Debug code can create unintended entry points in an application.	C, C++, Java	none

1

Appendix B Code Complexity Variations

In addition to having the capability to locate and identify source code weaknesses listed in Appendix A, a source code analysis tool must be able to find those weaknesses within all of these complex coding structures. A general list of these types of structures, adopted and modified from [MIT] is provided below. Some of the enumerated values are language specific (e.g. the use of pointers in C, C++), however, most are general types of constructs that exist across 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 or 2
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, class
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	1or 2
local control flow	type of control flow around weakness	if,switch,cond,goto/label,setjmp,longjmp, function pointer, recursion
loop complexity	component of loop that is complex	initialization, test, increment
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	same, inter-procedural, global,inter-file/inter-procedural, inter-file/global
taint	type of tainting to input data	argc/argv, environment variables, file or stdin, socket, process environment