Draft Special Publication 500-268

Source Code Security Analysis Tool Functional Specification Version 1.0

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

29 January, 2007

Michael Kass
Michael Koo

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 security analyzer. Because the majority of software security weaknesses today are introduced at the implementation phase, a specification that defines a "baseline" source code security analysis tool capability can help software professionals select a tool that will meet their software security 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 is working with the U.S. Department of Homeland Security to determine what the state of the art 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 the different classes software security assurance tools.

1.1 Purpose

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 vulnerability in the executable version of the application program. This document specifies a set of fundamental functional feature requirements for a source code security analysis tool, including a list of common code weaknesses that account for many of today's vulnerabilities in software.

This specification, together with the corresponding test plan and test suite, serves as a guide to measuring the capability of source code security analysis tools against this set of common weaknesses. In doing so, it is recognized that there are many quality tools that do not attempt to catch all of the weaknesses listed in this specification. The goal of this specification is not to prescribe the features and functions that all tools must have, but to identify some of the fundamental code weaknesses that greatly affect the security of software applications today, and provide a user of such tools with a way to measure if, and how well such a tool (or combination of tools) identifies these particular weaknesses.

Lastly, use of a tool (or toolkit) that complies with this specification does not guarantee "weakness-free" code. It does however provide a tool user with knowledge that their tool solution covers some of the most prevalent and highly exploitable security weaknesses that appear in source code today.

1.2 Scope

This specification is limited to 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 dynamically 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 security analysis and software assurance researchers

- Implementers and developers of source code security analysis tools
- Users and evaluators 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.

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 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.

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 security analysis tool examines software and analyzes 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 analysis 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 analysis 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 identify a construct as a vulnerability trades fewer false negatives for more false alarms and vice versa. The ideal would be a tool that identifies 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 identify a weakness, when in fact there is one present in the code.

false positive	Identification of a weakness by a tool, where there none exists.
false positive rate	Ratio of false positives to “true positives” reported by a tool for a given piece of code with known weaknesses.
weakness suppression system	A feature of source code security 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.
true positive	Correct identification of an actual weakness in source code by a tool.
weakness	A defect in a system that may (or may not) lead to a vulnerability.
weakness suppression system	A feature of source code security analysis tools that permits flagging of a line of code as “ignorable” by the tool in subsequent source code scans.

1
2
3
4

2 Functional Requirements

In this section we first give a high-level description of the functional requirements for source code security analysis tools, and 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 security analysis tool to be considered conformant with this specification. A source code security analysis tool (or tools) shall be able to (at a minimum):

- Identify a select set of software security weaknesses in source code.
- Generate a text analysis 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 security analysis tool(s) must be able to accomplish the tasks described in the mandatory requirements listed below. The tool(s) shall:

SCA-RM-1: Identify all of the code weaknesses listed in Appendix A. .

SCA-RM-2: Generate a text analysis of the code weaknesses that it identifies.

SCA-RM-3: Identify the weakness with a name semantically equivalent to those in Appendix A.

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

SCA-RM-5: Identify any weaknesses within the relevant the coding complexities listed in Appendix B.

SCA-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 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. Optionally, the tool(s) shall:

SCA-RO-1: Produce an XML-formatted report.
SCA-RO-2: Not identify a weakness instance which has been suppressed.
SCA-RO-3: Use the CWE name of the weakness it identifies.

3 References

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

[Kratkiewicz] 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 security analysis tool (or toolkit) 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 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 weakness is fairly easy for a malicious user to recognize and to exploit.

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

		data beyond the fixed memory boundary of a buffer in the heap portion of memory.		
Format string vulnerability	134	Tainted input is passed to the format string argument of a function, possibly exposing the stack content to a malicious user.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Improper Null Termination	170	Character data is passed to a string buffer without null termination and/or improper calculation of the appropriate location of the null terminator.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
API Abuse				
Heap Inspection	244	Memory is not cleansed prior to a call to the realloc() or free() function.	C, C++	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Often Misused: String Management	251	Tainted input and/or incorrect string length argument values used by *string manipulation functions* can produce a stack buffer overflow.	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 passed as an argument to a login function.	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	367	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	391	No action is taken after an error exception occurs in a	C, C++, Java	none

		program.		
Code Quality				
Memory leak	401	Memory is allocated in the program, but there is no logical path to a function that frees that memory.	C, C++	scope, address alias level, container, local control flow, loop structure
Unrestricted Critical Resource Lock	412	A resource is “deadlocked” in a way that results in a denial of service to users of the application.	C, C++, Java	asynchronous
Double Free	415	An attempt is made to free memory using a pointer that has previously been used in an earlier 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 the same memory location previously released by a call to the free() function.	C, C++	scope, address alias level, container, local control flow, loop structure, buffer address type
Uninitialized variable	457	A variable is created without assigning it a value. It is subsequently referenced in the program, causing potential undefined behavior or denial or service. .	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
Improper pointer subtraction	469	A pointer subtraction operation assumes contiguous memory allocation. If memory is not contiguous an erroneous memory reference and a potential denial of service	C, C++	scope, address alias level, container, local control flow, loop structure

		vulnerability exists.		
Null Dereference	476	Program input that results in a NULL dereference of a variable may expose a program to altered behavior that may permit exploitation of the program.	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	none

1

2

3

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 security analysis tool must be able to find those weaknesses within the complex coding structures relevant to that weakness. A general list of these types of structures, adopted and modified and extended from [Kratkiewicz] is provided below. Some of the complexities 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
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 or 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 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

1

2

3

4

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