Source Code Analyzer Tool Assessment Guide and Test Suite for the VVSG 2005 and the VVSG-NI

Version 1.0
April 1, 2009

This document and associated files have been prepared by the National Institute of Standards and Technology (NIST) and represent draft test materials for the Election Assistance Commission's Voluntary Voting System Guidelines. It is a preliminary draft and does not represent a consensus view or recommendation from NIST, nor does it represent any policy positions of NIST.

Preface

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the nation’s measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analysis to advance the development and productive use of information technology (IT).

ITL has developed this tool guide and accompanying tool tests to assist the U.S. Election Assistance Commission (EAC) accredited testing laboratories in assessing the effectiveness of source code analysis tools for potential use in determining source code conformance to the 2005 Voluntary Voting System Guidelines (VVSG) conventions. The Source Code Analysis Tool Guide and tool tests are part of a larger body of testing material that NIST is providing to test labs to augment their existing testing methods against the VVSG.

The Source Code Analysis Tool Guide provides a general background discussion of automated tools that test labs can use for source code analysis of voting systems, and instructions on how to use them with a collection of accompanying tool tests. The tool tests and test driver scripts provide a framework for calibrating tools to identify violations of coding conventions defined in 2005 VVSG.

This guide and tool tests is NOT a conformance test suite for determining a voting system’s conformance to the VVSG. Additionally, this guide and tool tests is NOT a conformance test suite for determining a source code analyzer’s conformance to any standard or specification. It provides useful information to help tool users choose and use those tools.

Use of this tool guide and tool tests are not mandatory for test labs. They are materials that a test lab can choose to use to enhance their existing tool calibration and code analysis methods. Its use can help in the selection of tools as well as provide useful information about a tool’s performance.

1 Introduction

1.1 Purpose

This guide and integrated tool tests are designed solely for the purpose of assisting test labs in assessing the capabilities of source code analysis tools that may be used during conformance testing to the VVSG. In particular, test labs may use software-testing tools to verify source code conformance against the VVSG 2005 or the VVSG Recommendations[1] coding convention requirements.

These software tools may be commercially available, in the public domain, or developed in-house by the test lab. Although a test lab need not use source code analysis tools, if tools are used, it is important that they work as purported and are capable of finding violations to the coding convention requirements. This guide and companion tool tests can help test labs make that determination.

The Source Code Analysis Tool Guide provides a general background discussion of some of the types of automated tools that test labs can use for source code analysis of voting systems, and instructions on how to use them with a collection of accompanying tool tests. The tool tests, and test driver scripts, provide a framework for calibrating tools to identify violations of coding conventions defined in the 2005 VVSG

Running a tool against source code files containing coding convention violations, and determining if the tool correctly identifies the type and location of those violations constitutes tool assessment. A positive assessment of a tool's coding convention identification capability provides confidence that the tool will find those violations in voting system software.

The goals of this guide include:

- Helping a test lab determine if a source code analysis tool will work as intended,
- Providing guidance to tool users on how to properly use the tools,
- Increasing public confidence in a test lab's ability to use appropriate tools when testing voting system software for conformance to the VVSG.

Please note that this is NOT a guide or test suite for determining a voting system's conformance to the VVSG. It is for tool calibration and assessment only.

1.2 Scope

There are many types of tools used to assure the quality of software today including requirements, architecture and design analysis, and static binary analysis tools. Additionally, dynamic testing tools are used to assure that software performs as intended

---

[1] VVSG Recommendations refers to the VVSG Recommendations to the EAC, December 2007.

and is free from defects and security vulnerabilities. This tool guide is limited to assessing the capabilities of the most commonly used source code analysis tools:

- Style-checkers,
- Compilers,
- Bug-checkers/security analyzers,
- Source code understanding tools.

Details of how these tools work, and the functionality that must be present in the tools to assess them are described in section 2.

1.3 Audience

This guide is primarily intended for Election Assistance Commission accredited test labs. Additionally, test labs and consultants performing state certifications of voting systems can benefit from this guide. It may also assist voting system manufacturers in their quality assurance (QA) efforts.

1.4 How to Use This Guide

This guide can be used as both a general reference for understanding source code analysis tools and their role in the VVSG conformance testing, and also for step-by-step instructions on how to assess a tool's capabilities.

At the higher level, this guide provides background information regarding what source code analysis is and the benefits that automated tools can offer in VVSG conformance evaluations. Additionally, the guide provides an overview of the four types of tools that can be employed and the functionality inherent in each type of tool

At the lower level (for test lab staff), this guide provides information and tool tests (in the form of sample source code) to help assess whether a source code analysis tool is appropriate for use in verifying source code conformance to VVSG coding convention requirements. The tool tests are documented with guidance on how to configure and use the tool, run the tool tests and analyze test results.

Beyond helping to determine if a tool is useful for source code analysis, the tool tests also provide a “tuning” mechanism for tools against an actual voting system that is “in house” for lab testing. Using the original tool tests as a guide, test lab staff can modify those tests or add new ones that are specifically designed to model the style and structure of that voting system's source code. Passing these “tailored” tests further improves the test lab's confidence in the tool, and what it will report when used on the actual voting system.

1.5 Related Documents

The U.S. Election Assistance Commission's 2005 Voluntary Voting System Guidelines (VVSG) specify the requirements that manufacturers of voting systems must implement in developing source code for voting systems. The following VVSG sections provided the technical requirements information needed to create this guide and tool tests:

- **VVSG 2005**
  - Volume I, Section 1.6, Conformance Clause,
  - Volume I, Section 5.2, Software Design and Coding Standards,
  - Volume II, Section 5, Software Testing.

- **VVSG Recommendations**
  - Part 3: Section 4.5.1.A and 4.5.1.B, Source Code Workmanship Requirements.

The VVSG contains the majority of coding convention requirements driving the creation of the tool tests. Some of the coding convention requirements in VVSG are also found in the VVSG Recommendations. This guide and tool tests covers that area of overlap, but do not address any new coding convention requirements in the VVSG Recommendations.

1.6 Structure of this Document

This guide is divided into three sections. In addition, supporting reference information is provided in Appendices A through D:

Section 1 is this introduction.

Section 2 provides general background information on the role of source code analysis tools in the software development process. Each of the four types of automated source code analysis tools is introduced. A short description of each type of tool is provided, along with a list of typical tool functions, including those necessary to use the tool tests.

Section 3 introduces the tool tests, describes the content of a test, and provides step-by-step instructions on the use of the tests.

Appendix A provides a complete list of all tool tests, cross-referenced against the tools for which they are applicable, and hyper-linked to the actual tests.

Appendix B provides an instructional end-to-end walk-through of procedures for performing a source code analysis tool test using one of the integrated tests as an example.

Appendix C lists the test metadata for the test walk-through.

Appendix D contains the source code file content used in the test walk-through.

2 Source Code Analysis Tools

The use of source code analysis tools can, through automated scanning, save test lab staff days or weeks of manual source code examination. In the case where the code consists of hundreds of thousands to millions of lines of code, automated source code analysis tools provide a repeatable, objective review that can quickly identify some of the common programming flaws that exist in today's software.

Because of their relative ease of use, scalability for large programs and maturity, source code analysis tools play a fundamental role in examining source code for compliance to coding conventions. Coding conventions are rules created by an individual software developer, a software development company, or a consortium that define style and structure in the writing of source code. Source code analysis tools can identify a breach of those conventions and find weaknesses in the code. Coding conventions can encompass practices that include source code formatting, commenting and naming conventions, as well as source code modularity, integrity and security. Source code analysis tools typically come "out of the box" with support for coding conventions that are generally accepted by the software development community. Additionally, most tools are extensible, permitting the tool user to expand and customize the number of coding conventions that a tool can identify and report on.

The VVSG defines specific coding conventions for voting system source code. Examples of VVSG coding conventions include:

- Each module shall be uniquely and mnemonically named, using names that differ by more than a single character. (VVSG 2005, Volume I, Section 5.2.3),
- No line of code exceeding 80 columns in width (VVSG 2005, Volume II, Section 5.4.2),
- Upon exit() at any point, presents a message to the user indicating the reason for the exit() (VVSG 2005, Volume II, Section 5.4.2).

Forty-three VVSG coding conventions are listed in Appendix A of this guide, each hyperlinked to one or more tool tests included with this guide. The tool tests are a collection of examples of source code that contain violations to these coding conventions.

2.1 Static Source Code Analysis Tool Functionality

Static source code analysis is a generalized term for examining source code for particular properties. The word "static" means that the code being examined is not actually executed (i.e. the code is not "dynamic" and running). The word "analysis" can have many meanings [1]. Analysis can be as simple as searching for particular strings of text in source code, and reporting them to the tool user. Analysis can also be complex, such as searching for bugs and/or security vulnerabilities in source code through dataflow and control flow analysis and property verification, or computing code quality metrics. Many tools limit themselves to performing a single function. General-purpose source code analysis tools typically perform a combination of these functions. Because tools vary in what they do, and how well they do it [2] [3], it is common for a tool user to employ a “toolbox” approach to source code analysis, utilizing multiple tools to help verify source code conformance to coding conventions.

2.2 Determining a Tool’s Suitability for the VVSG

The coding requirements specified in the VVSG pertain primarily to verification of correct coding style as opposed to code correctness, which is addressed in Section 2 of the VVSG. Because of the narrow scope of source code requirements, this document limits its scope to the discussion of four particular classes of tools that are suitable for this purpose:

- Source code style checkers,
- Compilers,
- Bug checkers and security checkers,
- Source code understanding tools.

To assist a test lab in determining conformance to VVSG coding conventions, source code style checkers, compilers, bug checkers and security checkers must*:*

- Correctly identify violations of one or more of the VVSG coding conventions listed in this guide in Appendix A,
- Correctly identify the coding convention violation location by file name and line number or by function name where appropriate.

Additionally, it is very helpful if a tool is flexible and allows for customization to VVSG requirements. Such tools should provide:

- Configuration options that permit disabling the search for coding convention violations that are irrelevant to VVSG,
- Extensibility that permits “customizing” a generic tool to identify VVSG-specific coding convention violations.

The fourth tool type, source code understanding tools, by themselves cannot identify a coding convention violation. This type of tool will always require human analysis for that purpose. That said, source code understanding tools are particularly useful for assisting a user in navigating source code and identifying coding convention violations that fully automated analysis tools do not find, and are therefore included in this guide.

2.2.1 Source Code Style Checkers

These tools focus on syntactic agreement of the source code with style-related coding conventions. The types of coding convention violations that they identify are primarily "readability" and "maintainability" related. The underlying search rules and engine for these tools are quite sophisticated. Common tool functions include:

- Enforcing indentation/whitespace rules for code and comments,
- Verifying code syntax against a particular API,
- Enforcing naming conventions,
- Enforcing encapsulation rules for object-oriented languages.

2.2.2 Compilers

As part of compiling source code, these tools report problems that will prevent compilation (in the form of compiler error messages) or that may produce unexpected runtime results (in the form of compiler warning messages). Compilers do not, as a rule, enforce coding conventions outside of the basic syntactic and semantic rules specified for that particular language; however some compilers (through command-line options) support the enforcement of a small set of programming style rules. A useful compiler function for improving code quality is type-checking analysis.

2.2.3 Bug Checkers and Security Analyzers

These tools identify source code problems of a deeper variety than those found by source code style checkers and compilers. Bug checkers and security analyzers identify flaws in source code that directly impact program predictability and security respectively.

Bug checkers find flaws in source code that may prevent the program from performing as intended. These flaws (often referred to as "bugs") can result in unexpected program behavior. Bugs are not necessarily security vulnerabilities unless they can be exploited in a malicious manner, but the unpredictable program behavior that they can produce is a serious availability and reliability issue. Some common bugs identified by this class of tool include:

- Uninitialized variables,
- Omitted break statements,
- Infinite loops,
- Null pointer dereferences.

Security checkers identify security-related coding flaws in source code. These types of flaws may have direct impact on the security of a program. Some common security-related flaws identified by this class of tool include:

- Potential buffer overflow,

- Race conditions,
- Presence of "debug" code in the final product.

2.2.4 Source Code Understanding Tools

These tools assist the tool user in understanding the structure of source code by generating a graphical, navigable and search-able representation of the code. They also generate commonly used code quality metrics that provide indicators of how well the code was constructed. Common tool functions include:

- Graphical control-flow and data-flow mapping and traversal,
- Keyword search through code by function, variable name or other property,
- Source code metrics generation, such as McCabe cyclomatic complexity and lines of code per module,
- Structural analysis (for possible code refactoring).

These tools serve as an aid to human analysis. By themselves, they cannot identify a coding convention violation. However as an aid to human analysis, they can greatly assist in helping the tool user identify areas of code in need of greater scrutiny and guide the tool user toward potential problems.

3 Source Code Analysis Tool Tests

The NIST source code analysis tool tests included with this guide consists of the set of tests and supporting documentation to help determine that a source code analysis tool can correctly locate and identify violations of the VVSG coding conventions. The tool tests reflect VVSG software design and coding requirements in the following areas:

- Integrity – requirements regarding self-modification at run time *(VVSG 2005, Volume I, Section 5.2.2),*
- Modularity – requirements defining module size and function encapsulation *((VVSG 2005, Volume I, Section 5.2.3),*
- Naming conventions – requirements for function and variable naming semantics, proper use keywords and name scoping *(VVSG 2005, Volume I, Section 5.2.5),*
- Control constructs – requirements for logic conventions such as loops and case statements and module entry/exit points *(VVSG 2005, Volume I, Section 5.2.4),*
- Comment conventions – self-documenting requirements regarding appropriate header files explaining function, I/O and revision history *(VVSG 2005, Volume I, Section 5.2.7),*
- Additional coding conventions – requirements that fall outside of the above categories *(VVSG 2005, Volume I, Section 5.2.6).*

There are multiple tests for each of these main areas, subdivided by the particular coding conventions that fall under these categories. The tool tests are small programs (i.e., source code files) that instantiate violations to specific coding practices, techniques, structure, etc.

The tool tests can assist a test lab by providing:

1) Basic tests to determine if a tool is, at a minimum, able to identify simple VVSG coding convention violations and not generate “false positive” reports,
2) An extensible testing framework that labs can augment with their own tests to:
    a. Better determine a tool’s capabilities against VVSG coding convention requirements,
    b. Emulate the coding style of a specific voting system to calibrate the tools specifically for that application,

Tool tests fall into two categories:

1) Tool tests containing a coding convention violation that a properly configured and calibrated tool can identify in source code with certainty, and without the need for further human analysis.
2) Tool-assisted human analysis tests, where a tool can identify a “potential” coding convention violation, but cannot verify it with one hundred percent certainty. These tests require additional human analysis to ascertain if a true coding convention violation exists or not.

The tool tests are written in 3 languages: C, C++ and Java. NIST recognizes that voting system software are written in more languages than these three, and that there are source code analysis tools for some of those other languages. The C, C++ and Java languages were chosen because they reflect the most commonly used programming languages, and have the greatest amount of tool support.

Tool Test Content

Each tool test contains at least one source code file (named Test.c, Test.cpp or Test.java) and an XML-encoded metadata file named metadata.xml that provides the necessary information to understand the test and run a tool against it.

The tool user uses the metadata to acquire an understanding of the tests, to see how tests are coded, and to identify the VVSG requirements that inspired the tests. The metadata also specifies any requirements for tool configuration or extensions necessary to run the tests. Additionally, the metadata is used to analyze and assess the tool’s output from executing the test. Specifically, the tool user compares the actual results reported from the tool to the expected results provided in the metadata.

The metadata includes:

- Test ID - A unique test identifier, based upon the directory path to the test, providing traceability for each test result,
- Test Description – A short description of the purpose of the test that provides the tool user with background information on what the test demonstrates about the tool's capability,
- Expected Results – Information the tool is expected to report after analyzing the test source code. This includes the location of the coding violation, providing:
  - The source file name(s), typically Test.c, Test.cpp or Test.java,
  - The line and column number of the violation,
  - The function or module name containing the violation.
- Test Design – The low-level details of how the source code is structured to produce the coding convention violation,
- Test Analysis – How a tool (or human being) would analyze the code to verify the presence of a coding convention violation. Includes circumstances that could complicate an analysis as well,
- Usage suggestions for appropriate tools that could assist in finding this particular coding convention violation, broken down by tool type, and containing recommendations for:
  - Tool configuration,
  - Tool extension.
- VVSG requirement identifier (by VVSG date, volume and section) – provides traceability to the VVSG coding requirement from which the test is derived.

See Appendix C for an example of this metadata file.

3.1 Installing and Using the Tool Tests

3.1.1 Installation

1. The tool tests are bundled in a ZIP file named sca_tool_guide_v1-0_and_tests.zip (for Windows platforms) and sca_tool_guide_v1-0_and_tests.tar.gz (for Unix systems). Extract all of the files from the archive into a location that is accessible to the tool(s) you wish to test. Please note that you must extract all of the files from the archive file to a disk directory for all the hyperlinks in this guide to work properly. Accessing this guide directly through WinZip or Microsoft Windows Explorer (the desktop file/directory browser) will result in unresolved tool test hyperlinks in Appendix A.

   a. The "nist_sca_tool_guide_and_tests" root directory contains this guide, named SCA_Tool_Assessment_Guide.doc, a directory named "xml" containing an xml schema and stylesheet for XML metadata validation and HTML rendering and the "tool tests" sub-directory containing the entire tool test collection. A README.txt file is also present, providing an overview of the content of the archive.

b. The “tool tests” directory contains 6 sub-directories corresponding to the six categories of coding conventions listed in section 3.
c. Within each category directory are sub-directories named for the particular coding convention violations that fall under that category.
d. Within each convention violation directory are subdirectories for each particular language that tests are written in (C, C++ and Java).
e. If more than one test is needed to verify whether a tool can identify a particular coding convention violation, then directories labeled 01, 02, 03 etc… exist for each individual test.
f. Within each numeric directory there are one or more source files, with a main file named Test.c, Test.cpp or Test.java for the appropriate language the test is written in.

As an example: the directory path to the first C-language tool test containing a violation of the array or string boundary limits coding convention is:

nist_sca_tool_guide_and_tests/
tool_tests/
integrity/
exceeding_array_string_bondaries/
c/
01/
Test.c

3.1.2 Using the Tests

The following steps guide the tool user through the process of selecting tests, understanding their meaning, configuring and running a tool against them. The workflow diagram in Figure 1 illustrates the process.

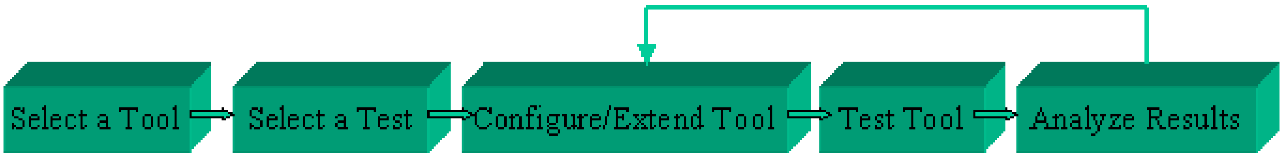

Figure 1 – Tool Test Workflow Diagram

Tool testing workflow may be a simple end-to-end process if the tool provides the functionality being assessed in the test “out of the box”. However, testing workflow may also be an iterative process. If the tool fails to correctly locate and identify the coding convention violation present in the test, the tool user may wish to re-configure and/or extend the tool and re-run the test in order to determine if the tool will be useful against that particular coding convention. This process can be repeated any number of times.

The following is an explanation of the steps in the tool test workflow.

Select a Tool

Determine which of the four tool types (source code style-checker, compiler, bug-checker/security analyzer, or code understanding tool) are applicable by consulting section 2.2 of this guide. Note that some tools are "hybrids", and have functions of more than one tool type; for example a code security analyzer may also have functions of a code understanding tool or a style checker may also have functions of a bug checker tool. Because of their broader functionality, hybrid tools can be run against a greater number of tests than "standalone" tools.

All of the tools, with the possible exception of some basic source code style checkers, will require the installation of a compiler in order to perform correctly. The compiler installation is necessary to satisfy the tool's search for source code headers and library functions referenced in the test code. If a C/C++ analysis tool is selected for testing, then a C/C++ compiler must be installed on the same machine. For java code analysis tools, a java compiler must be installed on the same machine.

Select a Test

1. Go to the Source Code Analysis Tool Tests table in Appendix A,
2. Click on a hyperlinked coding convention in column 1 that has an "X" in the column corresponding to the tool that you selected to test. This will take you to the root directory for all of the tests for that particular coding convention,
3. Move to the appropriate subdirectory (C, C++ or Java) based upon the programming language the tool can analyze,
4. One or more sub-directories, labeled 01, 02…. etc. will be present in this directory, corresponding to the number of tests for this coding convention,
5. Click on any sub-directory icon to change to that directory and examine a particular test,
6. Each test directory contains an XML metadata file, named metadata.xml and one or more source code test files. Each test file is labeled either Test.c, Test.cpp or Test.java, depending upon the programming language of the test,
7. Click on the metadata.xml file icon to display information about the test, including its description, and expected test results. It is best to view the XML metadata using a web browser,
8. Use the metadata provided in the XML file to:
    a. Get a basic description of the test (Test Description)
    b. Identify the directory location of the files for the test (Test ID)

   c. Understand what the tool should find in the code, and where it should find it (Expected Test Results)
   d. Understand the code-level details of the test (Test Design)
   e. Understand how a tool or human would analyze the code (Test Analysis)
   f. Trace the origin of the test back to the VVSG (Requirement ID)

Configure/Extend the Tool

If the tool does not find a particular coding convention violation “out of the box”, the metadata.xml file will also contain tool configuration and/or extension guidance in the “Tool Recommendations” section.

1. Examine the tool configuration information to learn what additional tool options need to be invoked to run this test.

2. Examine the extension recommendation to learn if tool extension may be required to run the test.

Test the tool

For tools with a command-line interface:

1. Open a shell window on the machine where the candidate tool is installed.
2. Change the working directory to the location of the test (from step 8 above).
3. Invoke the tool, including any required arguments needed by the tool (for example a compiler name is often required to invoke bug checking, security analyzers and source code understanding tools) and any additional arguments specified in the test configuration metadata. Use the name of the test file(s) Test.c, Test.cpp or Test.java as the target file for the tool.
4. Capture the output for future examination, typically by directing it to a file, for a permanent record.
5. To run the command-line tool in a “batch” mode, the tool user may wish to run the “run-tool.ps1” (for Windows) or “run-tool.sh” (for Linux) script located in “tool_tests” directory of the archive distribution. The tool user can incorporate the tool into the batch script by invoking the script and supplying the tool name, compiler, and programming language. The batch scripts are self-documented with comments that explain how to run them.

For tools with a graphical user interface:

1. Invoke the tool from the desktop environment.
2. Configure the tool through its graphical interface, providing any necessary project name, workspace location, compiler name or other required setup information

required for the tool to run. This information can be found in the tool's documentation.

3. Open the main test file Test.c, Test.cpp or Test.java through the user interface.
4. Run the automated analysis. For code understanding tools, perform the manual analysis described in the "Test Analysis" portion of metadata.xml file.

Analyze Results

Compare the tool report against the expected test result information provided in the metadata.xml file. Please note that tools may not always report a coding convention violation at the same source code line number as that specified in the test metadata file. This particularly applies to coding convention verification that requires dataflow analysis (such as a buffer overflow). This is not considered a tool error. If the tool reports the same coding convention violation on a line other than the one specified in the test metadata, human analysis can verify if the tool still correctly identified the violation.

1. For automated reporting tools (e.g. style checkers, bug checkers, compilers and security analyzers), verify that the tool reports the coding convention violation named in the metadata file (from step 8 of the Select a Test section above) using a semantically equivalent name, along with its location, including file name and line number, and for code metrics, module name.
2. For human analysis tools (e.g. code understanding tools), verify that you were able to identify the coding convention violation in its correct file, line number and for code metrics, module name.

Modify Configuration/Extension of Tool

If you determine that the tool was not properly configured and/or extended for this test, you may revisit the tool configuration/extension step and run the tool again against this test.

4 References

[1] Brian Chess, Jacob West, “Secure Programming with Static Analysis”, Addison-Wesley Software Security Series, 2007

[2] Misha Zitser, Richard Lippmann, and Tim Leek, “Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code**”**, Proc. FSE-12, ACM SIGSOFT, 2004.

[3] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster, “A Comparison of Bug Finding Tools for Java” - The 15th IEEE International Symposium on Software Reliability Engineering ISSRE'04). Saint-Malo, Bretagne, France. November 2004.

Appendix A – VVSG Coding Convention and Tool Test List

Table 1 below provides a list of the VVSG source coding conventions. Each coding convention name is hyperlinked (via the, “caliper” icon) to the root directory for the tests associated with that convention. Please note that this guide must reside in the root “nist_sca_tool_guide_and_tests” directory of the archive distribution and that the “tool tests” directory must be directory below it for the hyperlinks in this table to work properly.

A corresponding “checked box” indicates which type of tool (or tools) may assist test labs in identifying a violation of that coding convention. In some cases, there are coding conventions that no tool can verify conformance to. Those coding conventions require human analysis for verification and therefore no tool tests can be written for them.

This list provides a recommended starting point for selecting a tool and calibrating it against a coding convention. Keep in mind that today’s tools are increasingly “hybrid” in design, and often possess the capabilities of more than one tool type. For example, security checkers may also perform some of the same code checks as bug checkers. And bug checkers may perform some of the same code checks as style checkers. This table takes a “least common denominator” approach, listing the tool most appropriate to start with to verify a coding convention. For most of the VVSG coding conventions, the “style checking” tool is most appropriate. For some, a higher-complexity tool is indicated.

A detailed discussion of how to use or configure or extend a tool to identify the coding convention violation is included in the XML documentation file (named metadata.xml) that accompanies each tool test.

VVSG Coding Convention (abridged for brevity with full requirement reference provided)	**Tool Types**			
	Style Checker	**Compiler**	**Bugs/ Security Checkers**	**Understanding Tools**
Software associated with the logical and numerical operations of vote data shall use a high level programming language such as Pascal, Visual Basic, Java, C and C++. The requirement for the use of high-level language for logical operations does not preclude the use of assembly language for	X			

hardware-related segments, such as device controllers and handler programs. VVSG 2005: 1:5.2.1				
Self-modifying code is prohibited, except under the provisions outlined in Subsection 7.4. VVSG 2005:1:5.2.2	Languages that are self-modifying (COBOL, Lisp, SNOBOL, assembly language) are not covered by this guide and tests.			
Dynamically loaded code is prohibited, except under the security provisions outlined in Subsection 7.4. VVSG 2005:1: 5.2.2	x			
Interpreted code is prohibited, except under the provisions of Subsection 7.4. VVSG 2005:1:5.2.2	Requires manual inspection of code submission for presence of interpreted language source code files.			
The software shall provide controls to prevent accidental or deliberate attempts to replace executable code through unbounded arrays or strings (includes buffers used to move data) .VVSG 2005:1:5.2.2			x	
The software shall provide controls to prevent accidental or deliberate attempts to replace executable code via pointer variables. VVSG 2005:1:5.2.2			x	
The software shall provide controls to prevent accidental or deliberate attempts to replace executable code via dynamic memory allocation and management. VVSG 2005:1:5.2.2			x	
Each module shall have a specific function that can be tested and verified. VVSG 2005:1:5.2.3.a				X
Each module shall have a single entry point, for normal process flow. VVSG 2005:1:5.2.3.e	This guide and tests do not cover languages that permit multiple entry points (Fortran, COBOL) at this time.			
Each module shall have a single exit point, for normal	X			

process flow. VVSG 2005:1:5.2.3.e				
The exception for the exit point is where a problem is so severe that execution cannot be resumed. In this case, the design must explicitly protect all recorded votes and audit log information and must implement formal exception handlers provided by the language.	X			
Voting system software shall use the control constructs identified in this section as follows: sequence, if-then-else, do-while, do-until, case, general loop, including special case "for" loop (or their equivalents, as defined and provided by the vendor). No other constructs shall be used to control program logic and execution. VVSG 2005:1:5.2.4.a	X			
Operator intervention or logic that evaluates received or stored data shall not re-direct program control within a program routine. VVSG 2005:1:5.2.4	Because of the unique nature of each voting system implementation, no generic tool tests can be written for this requirement.			
Do-While (false) constructs are prohibited. VVSG 2005:1:5.2.4.a.iii	X			
Intentional exceptions (used as gotos) are prohibited. VVSG 2005:1:5.2.4.a.iii	It is beyond the scope of the tools listed here to determine if an exception is thrown for valid reason or not. This requires human inspection of every exception verify this coding convention.			
Names used in code and in documentation shall be consistent. VVSG 2005:1:5.2.5.b	Beyond the capabilities of the tools listed here. Documentation examination tools are not part of this guide.			
[Object, function, procedure, and variable names] shall be chosen to enhance the readability and intelligibility of the program. Names shall be selected so that their parts of speech represent	This is beyond the capabilities of tools listed here. It requires human knowledge of the meaning of the names and their function in the application.			

their use, such as nouns to represent objects and verbs functions. VVSG 2005:1:5.2.5.c				
[Object, function, procedure and variable names] shall differ by more than a single character. VVSG 2005:1:5.2.5.c	X			
All single-character [object, function, procedure and variable names] are forbidden except those for variables used as loop counters. VVSG 2005:1:5.2.5.c	X			
Language keywords shall not be used as names of objects, functions, procedures, variables or in any manner not consistent with the design of the language. VVSG 2005:1:5.2.5.d	C, C++ and Java compilers do not permit the use of keywords as objects, functions, procedures or variables, so there are no tests against this coding convention. Other languages that do permit this (e.g. Fortran) are not covered by these tests.			
All modules shall contain headers. For small modules of 10 lines or less, the header may be limited to identification of unit and revision number. VVSG 2005:1:5.2.7	No generic tool tests can be written for this coding convention, as the commenting style for headers varies from application to application.			
Header comments shall provide the following information: the purpose of the unit and how it works, other units called and the calling sequence, a description of input parameters and outputs, file references by name and method of access (i.e., read, write, modify or append), global variables used and date of creation and a revision record	No generic tool tests can be written for this coding convention, as the commenting style for headers varies from application to application.			
Uses uniform calling sequences. All parameters shall either be validated for type and range on entry into each unit, or the unit comments shall explicitly identify the type and range for the reference of the programmer or tester. Validation may be	This convention goes beyond implicit compiler type validation of parameters. It requires human analysis to verify that parameters are validated against what is specified in unit comments.			

performed implicitly by the compiler or explicitly by the programmer VVSG 2005:2:5.4.2				
Has the return explicitly defined for callable units such as functions or procedures (do not drop through by default) for C based languages and others to which this applies. VVSG 2005:2:5.4.2	X			
Where the return is only expected to return a successful value, the C convention of returning zero shall be used, or the use of another code justified in the comments. VVSG 2005:1:5.4.2	This requires human analysis to verify if a function is returning a “success” indicator, or an integer value to be used for another purpose. Automated tools cannot make this determination.			
Does not use macros that contain returns or pass control beyond the next statement. VVSG 2005:2:5.4.2	X			
For those languages supporting case statements, has a default choice explicitly defined. VVSG 2005:2:5.4.2	X			
Provides controls to prevent any vote counter from overflowing. VVSG 2005:2:5.4.2			X	
Code is indented consistently and clearly to indicate logical levels. VVSG 2005:2:5.4.2	X			
Excluding code generated by commercial generators, no more than 50% of modules exceed 60 executable lines, no more than 5% of modules exceed 120 executable lines, no modules exceed 240 executable lines VVSG 2005:2:5.4.2	X			
Where code generators are used, the source file segments provided by the code generators should be	This is beyond the verification capability of the tools discussed here.			

marked as such with comments defining the logic invoked. VVSG 2005:2:5.4.2				
Has no line of code exceeding 80 columns in width (including comments and tab expansions) without justification VVSG 2005:2:5.4.2	X			
Contains no more than one executable statement or no more than one flow control statement for each line of source code. VVSG 2005:2:5.4.2	X			
Avoids mixed-mode operations. If mixed mode usage is necessary, then all uses shall be identified and clearly explained by comments. VVSG 2005:2:5.4.2	X			
All normal status messages shall be self-explanatory and shall not require the operator to perform a lookup to interpret them. VVSG 2005:2:5.4.2	This requires human analysis to determine if status message is self-explanatory. Automated tools cannot make this determination.			
All error status messages shall be self-explanatory and shall not require the operator to perform a lookup to interpret them, except for error messages that require resolution by a trained technician. VVSG 2005:2:5.4.2	Requires human analysis to determine if status message is self-explanatory. Automated tools cannot make this determination.			
If an uncorrected error occurs, and the unit must return without correctly completing its objective, a non-zero return value shall be given even if there is no expectation of testing the return. VVSG 2005:2:5.4.2	This requires human analysis to determine if an uncorrected error has occurred (vs. a corrected error) and whether a zero or non-zero value should be returned.			
References variables by fewer than five levels of indirection (i.e., a.b.c.d or a[b].c->d) VVSG 2005:2:5.4.2	X			
Module has functions with fewer than six levels of indented	X			

scope. VVSG 2005:2:5.4.2				
Initializes every variable upon declaration where permitted. VVSG 2005:2:5.4.2	X			
Has all constants other than 0 and 1 defined or enumerated, or shall have a comment which clearly explains what each constant means in the context of its use. VVSG 2005:2:5.4.2	X			
Only contains the minimum implementation of the “a = b ? c : d” syntax. Expansions such as “j=a?(b?c:d):e;” are prohibited. VVSG 2005:2:5.4.2	X			
Has all assert statements coded such that they are absent from a production compilation. VVSG 2005:2:5.4.2	X			

Table 1 – VVSG Coding Convention and Tool Test List

Appendix B – Tool Test Walkthrough

This section walks the reader through the process of assessing a “candidate” source code security analyzer tool that a test lab may wish to include in its toolbox for verifying conformance to VVSG coding conventions.

1. A test lab wants to assess Tool A’s potential for use in identifying violations of VVSG coding conventions in source code. The tool documentation claims that it finds many common security flaws in source code, including buffer overflows in both C and C++ programming languages. The tool reports what it finds in a text file, providing the name of the coding flaw, the file name and module containing it, and the line number where it reports the flaw.

2. The tool user installs the NIST Source Code Analysis Tool Assessment Guide and companion tool tests from the downloaded archive. The tool user consults section 2.2 of the guide and determines through comparison with the tool documentation that that the tool fits the functional profile of a “security analyzer”.

3. Consulting Appendix A of this guide (an excerpt is illustrated below in table 2), the tool user notes that one test that can be used to verify whether the tool may be useful for VVSG source code conformance testing.

VVSG Coding Convention	Style	Compiler	Bugs/Security	Understanding
The software shall provide controls to prevent accidental or deliberate attempts to replace executable code through unbounded arrays or strings (includes buffers used to move data) .VVSG 2005:1:5.2.2			x	

Table 2 – Excerpt from Source Code Analysis Tool Test List

4. Clicking on the “caliper” hyperlink in the test table takes the tool user to the root directory of all of the tests for this coding convention. The tool can analyze both C and C++ source code, and the tool user chooses to first run the C-language tests. Clicking on the “c” directory brings up a list of subdirectories 01 through 05, corresponding to tests 1 through 5. The tool user selects the first test directory, 01, and is presented with the test file itself, labeled “Test.c” and the accompanying metadata file “metadata.xml”.

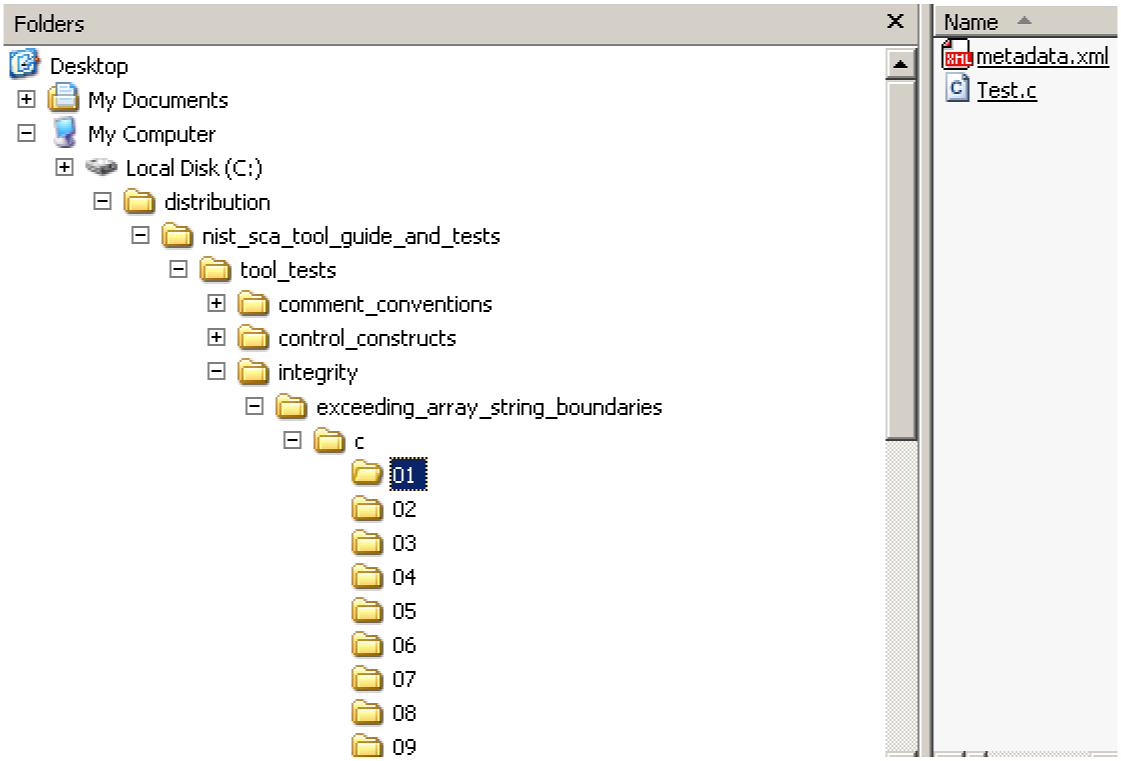

Figure 2 – Graphical view of the tool_tests directory structure

5. Opening the metadata.xml file (using a web browser) reveals all of the information necessary to understand the test and run the tool against it.

- The Test Description tells the tool user that the purpose of this test is to verify that the tool can at a minimum identify a simple character array overflow.
- The Expected Results indicate that after executing the test, the tool users should find a coding convention violation semantically equivalent to the name "exceeding array or string boundaries" in the file Test.c at line 25 in the module "main".
- The Test Design allows the tool user to examine and understand the test code. In this test, a character array named "char_array" is overwritten through an iterative loop that copies an "A" character to each element in the array, beginning at index 0 through 255. However, the array is only defined for a length of MAXLEN, or 40 characters.

6. Reading the "Test Analysis" information, the tool user learns:

- The tool must recognize that stack memory can be overwritten when the loop counter exceeds the declared length of a character array, and identify the line number where the actual overflow can take place.
- Other factors may influence a tool's ability to find this flaw in other source code.

7. Examining the "Tool Recommendations" section, the tool user can determine if any additional tool-specific configuration or extension is necessary to run the test:

- The “Configuration” section states that no tool configuration measures are necessary prior to running the tool against the test.
- The “Extension” section states that no tool extension measures are necessary because this type of tool generally comes with the capability to find a buffer overflow “out of the box”.

8. The tool user runs the tool. The report from Tool A is displayed below:

```
Coding Flaw Name : Buffer Overflow
Location: File = Test.c,  line = 25
```

These results indicate that the source code security analysis tool found a “buffer overflow”, a semantically equivalent name to the “exceeds array boundaries” coding convention violation. The tool found the violation in the file Test.c, at line 25. Other information may be present in a report, but is not required for evaluating the tool.

9. Examining the actual source code in the file Test.c, the tool user verifies that this is in fact where the array boundary is exceeded at line 25, as annotated with the comment /*BAD*/ on that line. The tool user can then compare the tool report against the “expected results” described in step 5

10. If the tool user thinks the tool was not adequately configured and/or extended to run this test, the tool user may revisit the configuration and extension information, possibly revising the tool setup, and run the test again. The tool user can iterate through steps 7-9 as necessary,

11. Noting the VVSG requirement ID: VVSG 2005:Vol 1:5.2.2, the tool user can go to that VVSG and examine the testing requirement from which this test is derived.

Appendix C – Metadata for Tool Tests

Below is the metadata for one of the tool tests distributed with this guide. The test is used in the walk-through described in Appendix B.

Test Identifier

Integrity/exceeding_array_string_bondaries/c/01

Test Description

This test case is a simple example of a buffer overflow that any generalized bug checking or security analysis tool should correctly identify. It can be used to quickly determine if the tool can identify a simple buffer overflow at all.

Expected Test Results

File: path = Test.c

Coding Convention Violation: name = exceeding array string boundaries, filename= Test.c, line = 25

Test Design:

This is an example of a buffer overflow, in which data is written to a character array beyond its bounds through direct user input from the command line. A character array named char_array receives 255 'A' characters via a loop. The size of char_array is defined by the MAXSIZE macro definition, which is only 40. Execution of this program will result in termination of the program and the display of a “stack overflow” message.

Test Analysis:

This test is simple character array overflow introduced by the program itself, without human input. It is simple in the sense that coding constructs that could impede a tool's ability to locate the buffer overflow are not present in this test. A tool should recognize that no check is made by the program to verify that character array is not written to beyond its 40-character limit and that a stack memory overflow will occur.

Complex coding constructs such as pointer aliasing or inter-procedural dataflow can confuse a tool and result in a "false negative" report, indicating that it did not find the buffer overflow. To verify that your tool can find buffer overflow in more complex source code, run the tool against all of the other buffer overflow examples in this directory.

Tools can also generate "false positive" reports, meaning that no real potential for a buffer overflow exists, but the tool reports that there may be one present in the code. This requires human inspection to verify if in fact a buffer overflow exists.

Almost all bug checkers and source code security analysis tools generate false-positive reports, and a tool should not necessarily be dismissed because it generates them. A test laboratory must determine when the percentage of false-positive to true-positive (actual) violations generated by a tool is acceptable based upon people available to investigate them. .

Run the other tests for this coding convention to see if the tool you are assessing generates "false negative" or "false positive" reports.

Tool Recommendations:

Bug Checker or Security Analyzer

Configuration:

No special configuration is needed for this class of tool to detect buffer overflows.

Extension:

Custom extension for buffer overflows is typically not necessary in today's generalized security analysis tools. The tools generally support finding this coding flaw "out of the box". If it does not, then tool extension may be necessary. However, extension of a tool to find this particular flaw would be a non-trivial task. A better choice would be to find a tool that identifies this flaw as a default part of its core functionality.

Requirement ID: VVSG 2005:Vol 1:5.2.2

Appendix D - Test Source Code

Below is the source code for one of the tool tests that accompany this guide. The code is used in the tool test walk-through in Appendix B. The source code is typical of the size and complexity of the tests in this suite. A single coding convention violation (a buffer overflow) is present in the file, and highlighted with a /* BAD */ comment to help identify it.

```
/*
* This test is part of the NIST Source Code Analysis Tool Assessment Test Collection
* Test ID: tool_tests/integrity/exceeding_array_string_bondaries/c/01
*
*This software was developed at the National Institute of Standards and
* Technology by employees of the Federal Government in the course of their
* official duties. Pursuant to title 17 Section 105 of the United States
* Code this software is not subject to copyright protection and is in the
* public domain. NIST assumes no responsibility whatsoever for its use by
* other parties, and makes no guarantees, expressed or implied, about its
* quality, reliability, or any other characteristic.
*
*/

#defineMAXSIZE			40

int
main()
{
  char char_array[MAXSIZE];
  int i;

  for( i = 0; i < 255; i++)
    char_array[i] = 'A';		/*BAD*/

	return 0;
}
```