# Creating a Baseline Specification for Source Code (Security) Analysis Tools

OMG Software Assurance Workshop
March 6, 2007

Michael Kass,  NIST Information Technology Laboratory

http://samate.nist.gov
michael.kass@nist.gov

# Source Code Security Analysis Tool Functional Specification

- Introduction
  - Background on DHS/NIST SAMATE project
- How to use this specification
- Tool functional requirements
  - Finding weaknesses
  - CWE compatibility
  - Code complexities
- Evaluating tools against this specification
  - Other tool evaluation efforts
  - SAMATE SRD repository
- Tool metrics
  - Issues in defining a "general" metric
- Summary of Work to Date and Future Plans

## SAMATE Background

- **Software Assurance Metrics and Tool Evaluation project began work in 2005, sponsored by DHS**
- ***NIST is:***
  - **A non-regulatory agency in Dept. of Commerce with 3,000 employees + adjuncts**
  - **Over a century of experience in standards and measurement**
  - **Information Technology Laboratory**
    - **Federal Information Processing Standards (FIPS)**
      - **Security Requirements for Cryptographic Modules, EDI, SHS, DSS**
    - **NIST Special Publications (SP 800 Series Computer Security Guidelines)**
    - **National Voluntary Lab Accreditation Program (NVLAP)**
      - **Common Criteria and Crypto Module testing labs**
    - **Software Diagnostics and Conformance Testing Division (SDCT)**
      - **Develops test plans and test suites against IT standards (SQL, Posix, PHIGS, XML…others..)**

# SAMATE Background

- Workshop: Survey the State of the Art in SwA tools (August ’05)
  - Classify SwA tools across SDLC
  - Choose a class of SwA tool to develop a functional specification
  - Enlisted volunteers to help:
    - Review Tool Functional Specifications
      - Source  Code Analysis Tools
      - Web Application Scanners
    - Contribute test cases against the specifications to the SAMATE Reference Dataset (SRD)

## SAMATE Products

SwA Tool Functional Specifications

Tool Test Suites

Tool Metrics

----------------------------------------

Tool Evaluations

# Specifications

- NIST SP-500-268 “Source Code Security Analysis Tool Functional Specification”
  - Defines functional requirements for mandatory and optional features
  - Links functional tool requirements (finding weaknesses) to the Common Weakness Enumeration (CWE)
  - Defines code complexities tools must handle
- Test suite for this specification
  - discrete tests with known weaknesses
  - real world application examples
  - false-positive tests
  - auto-generated tests based on formal CWE definitions

## How to Use This Specification

- Audience
  - Tool users
  - Tool developers
  - Researchers
- Scope
  - Source code analysis only ( not bytecode, binary, or dynamic analysis )
  - Current languages are C, C++ and Java

## Context for this Document

- **The Specification:**
  - Is a definition of a baseline capability among general-purpose, production-type source code security analysis tools
  - Addresses some of today's more common, detectable, high severity SwA problems
  - Is not exhaustive for all weaknesses
  - Is not prescriptive for all tools
    - e.g. special purpose tools that address small problem space
  - Does not guarantee "weakness-free" code if a tool complies to this specification
  - Will evolve
    - With tool capabilities
    - With new weaknesses

# State of Source Code Analysis Tools Today

- Generic in design
- Customizable
- Little overlap in CWE coverage between tools
  - NSA Pilot Tool Evaluation Results
  - MITRE Survey of Tool Claims against CWE
- Toolbox approach is needed to cover a weaknesses defined in specification

## Key Technical Issues

- What is a “baseline capability” for a general purpose source code security analysis tool (or solution of tools)?
  - What functions should these tools share?
  - What weaknesses should they catch?
  - What coding structures should they be able to handle?
- How do we deal with false positives?

# Tool Functional Requirements

- Mandatory Features: The tool(s) shall
  - **SCA-RM-1:** Identify the code weaknesses listed in Appendix A.
  - **SCA-RM-2:** Generate a text analysis of code weaknesses that it identifies.
  - **SCA-RM-3:** Shall 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 coding complexities listed in Appendix B.
  - **SCA-RM-6:** Have an acceptably low "false-positive" rate.

## Code Weakness Selection Criteria

- Found in code today
  - Listed in National Vulnerability Database
- “Catch-able” by tools
  - According to tool vendor claims
- High likelihood of exploitation
  - As defined by CWE

# Code Weaknesses (and CWE ID)

Basic XSS 80

OS Command Injection 78

SQL Injection 89

Stack Overflow 121

Heap Overflow 122

Format String Vulnerability 134

Improper Null Termination 170

Heap Inspection 244

String Management 251

Hard-Coded Password 259

TOCTOU race condition 367

Unchecked Error Condition 391

Memory Leak 401

Unrestricted Critical Rsrc Lock 412

Double Free 415

Use After Free 416

Un-initialized Variable 457

Unintended Pointer Scaling 468

Null Dereference 476

Leftover Debug Code 489

## Code Complexities

- address alias level
- array address complexity
- array length/limit complexity
- asynchronous
- buffer address type
- data type
- index alias level
- local control flow
- loop complexity
- loop iteration type
- loop structure
- memory access
- memory location
- scope
- taint

# Combining Weaknesses with Complexities

index complexity = constant , secondary control flow = if, loop structure = non-standard for, scope = inter-procedural, local control flow = function pointer

```
int main(int argc, char *argv[])
{
  void (*fptr)(char *);
  int test_value;
  int inc_value;
  int loop_counter;
  char buf[10];

  test_value = 10;
  inc_value = 10 - (10 - 1);

  for(loop_counter = 0; ; loop_counter += inc_value)
  {
    if (loop_counter > test_value) break;
fptr = function1;
fptr(buf);
}
  return 0;
}
```

```
void function1(char
*buf)
{
 /*  BAD  */
   buf[10] = 'A';     ****
code weakness ***
}
```

# Evaluating Tools Against this Specification

- SAMATE Reference Dataset (SRD)
  - An online repository of test cases for SwA Tools ( not just source code security analysis tools )
  - Contributors include:
    - NIST SAMATE Team
    - Tool Vendors (Fortify Software, Klocwork, Praxis)
    - Research/Academia (MIT, R. Seacord, DRDC)
    - KDM Analytics – Auto-generated test cases from formalized CWE definitions
  - SAMATE Test Plans and Test Suites
  - Many possible lab testing models (TBD)
    - NIST, NVLAP, 3rd Party Lab Accreditation (e.g ITEA), Self-test

## Tool Metrics

- The challenge of creating a metric that combines:
  - True positives
  - False positives
  - False negatives
  - Unsupported weaknesses

# Summary of Work to Date and and Future Plans

- Source Code Security Analysis Tool Functional Specification, NIST SP 500-268, is online at *http://samate.nist.gov/index.php/Source_Code_Security_Analysis*

- SAMATE Reference Dataset
  - ~1600 Test Cases (C, C++, Java)
  - Source Code Security Analysis Tool Test Plan
  - Source Code Security Analysis Tool Test Suite

- Future Plans
  - Expand test suite beyond current languages
  - Update specification as tool capabilities align and new weaknesses need to be added to the "baseline"
  - Define a useful and generally accepted metric for tool comparison