The SAMATE Project and Its Relation to SCADA

Electricity Advanced Sensing and Control Conference

January 11, 2006

Michael Kass
Information Technology Laboratory
National Institute of Standards and Technology
http://samate.nist.gov
michael.kass@nist.gov

Outline

- Brief background on the SAMATE project
- Overview of work done to date
- How does this relate to SCADA?
- Next steps

The Questions

- Do software assurance tools work as they should?
- Do they really find vulnerabilities and catch bugs? How much assurance does running the tool provide?
- Software Assurance tools should be …
  - Tested: accurate and reliable
  - Peer reviewed
  - Generally accepted

DHS/NIST Address the Issues

- Assess current tools and methods in order to identify deficiencies which can lead to software product failures and vulnerabilities
- Identify gaps in tools and methods, and suggest areas of further research DHS sponsors NIST to:
- Develop metrics for the effectiveness of SwA tools, and to identify deficiencies in software assurance methods and practices
- NIST creates the Software Assurance Metrics and Tool Evaluation Project (SAMATE)

How does this relate to SCADA?

- SwA tools are increasingly being used to provide an argument for an applications software assurance
- SwA tools are used to verify/validate requirements, design, implementation and deployment of software systems
- SCADA systems can be vulnerable to the same types of attack as any other application, particularly networked applications

Work to Date

- 2 Workshops (August and November 2005)
- Goals of workshops:
  - Define the State of the Art (SOA) and State of the Practice in SwA tools today
  - Begin work to enable comparison and measurement of SwA tool capabilities
- Attendees included:
  - SwA tool vendors, private and academic researchers, SwA tool users, SwA consultants, government (DHS, NSA, NIST)

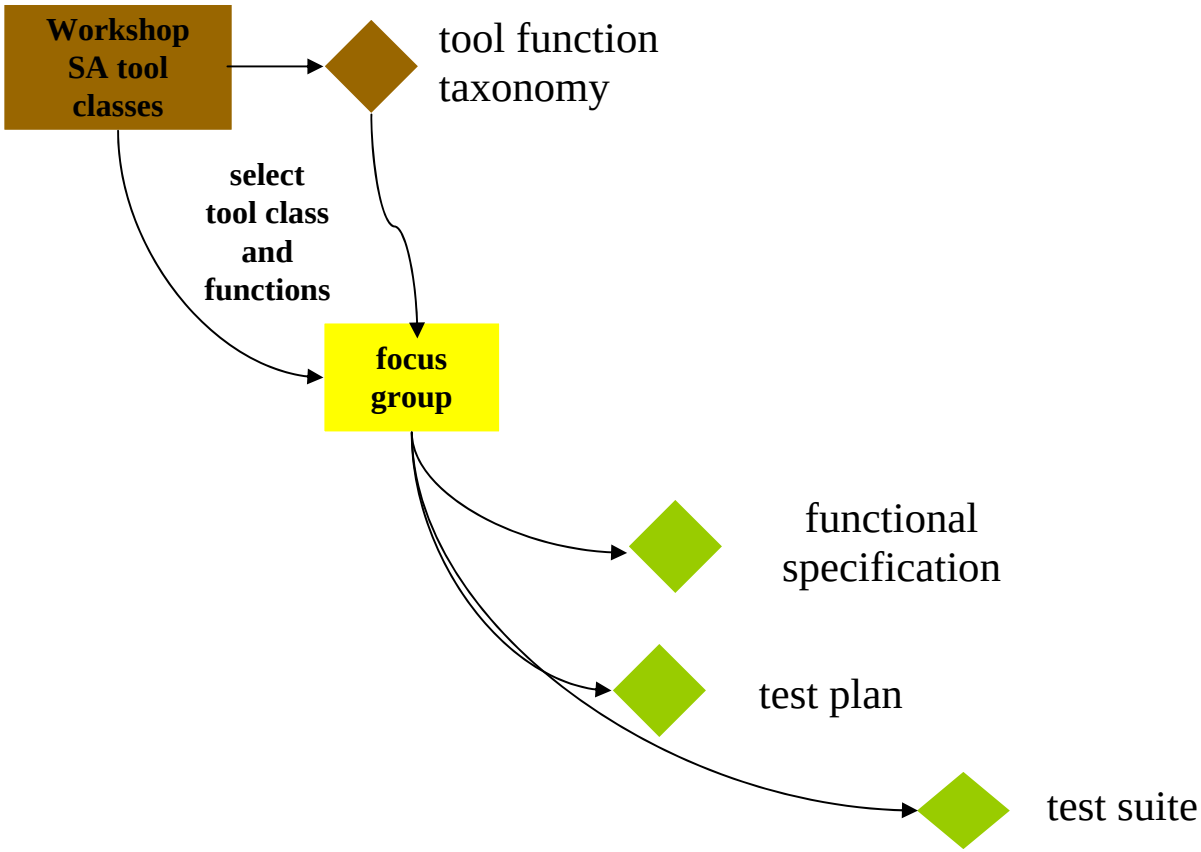
Workshop Goals

- Develop a common taxonomy of Software Assurance (SwA) tools and their functions
- Define a “common/agreed-upon” taxonomy of software security flaws
- Develop an open reference dataset of test material for SwA tools
- Develop functional specifications for SwA tools
- Define common metrics for applications and SwA tools
- Identify gaps in capabilities of today’s tools

A SwA Tool/Function Taxonomy

- Provides a common reference/classification of functions for evaluation of a tool's effectiveness
- A taxonomy is a "first step" in defining a tool functional specification
- A tool specification becomes basis for creating tool reference dataset/benchmarks
- A reference dataset for comparing SA tools of the same class permits an "apples to apples" comparison between tools

Tool Taxonomy and Follow-on Products



SwA Tool Taxonomy (high-level)

”External” Tools

- Network Scanners
- Web Application Scanners
- Web Services Scanners
- Dynamic Analysis Tools

Much of the information in the Software Security Tool Taxonomy is derived from the DISA Application Security Assessment Tool Survey, July 2004, to be published as a DISA STIG. Additionally, Secure Software’s CLASP Reference Guide V1.1 Training Manual was used to compile a list of common software vulnerability root causes.

SwA Tool Taxonomy (cont.)

"**Internal" Tools**

- Compilers
- Software Requirements Verification
- Software Design/Model Verification
- ***Source Code Scanners***
- Byte Code Scanners
- Binary Code Scanners
- Database Scanners

Why Source Code Scanners?

- Implementation errors account for majority of security vulnerabilities in software
- Source code scanners find buffer overflows... which account for over 50% of security vulnerabilities
- Source code scanners can be used during the implementation phase of the SDLC as well as for examining legacy code that has no other artifacts to examine
- Source code scanners are more reliable than binary scanners (you can actually examine the source code)
- Software of “unknown pedigree” may only have source code available to audit

Source Code Scanner Functions:

- Identify vulnerabilities in code
- Filter results (adjust false-positives)
- Categorize vulnerabilities (by severity, taxonomy)
- Generate code metrics

Scanner Function: Identify Vulnerabilities in Code

The ability to identify flaws that may lead to vulnerability varies from tool to tool, and the underlying techniques that they use. One of the first steps in comparing tools of the same class is to determine what software flaws all tools of that class should be able to identify.

- Identify range and type errors
- Identify environmental problems
- Identify synchronization and timing errors
- Identify protocol errors
- Identify general logic errors

The above hierarchical classification of software flaw root causes is taken from Secure Software's CLASP V1.1 Training Manual

Defining a Common Software Flaw Taxonomy

- There are multiple taxonomies to choose from/integrate (CVE Plover, CLASP, Fortify Software..others) We need to integrate them into one common, agreed-upon taxonomy
- Some of the flaws in the taxonomy cannot be identified by tools today
- Some flaws have not been seen in real code yet
- A flaw taxonomy must cover the entire SDLC (flaws introduced during requirements, design, implementation, deployment)
- A taxonomy should also contain axes/views such as “remediation” or “point of introduction in SDLC” (some of which already exist in CLASP)
- Volunteers to help with the flaw taxonomy include: Cigital, Mitre, Ounce Labs, Klockwork Inc., Secure Software Inc., Fortify Software, OWASP

Creating a Reference Dataset

- A reference dataset for static source/binary code analysis should reside in an accessible network repository
- A reference dataset for dynamic analysis tools should reside in a network testbed
- We need to define control/access to a network testbed and network repository
- Dynamic testing of legacy binary software on “new” platforms is very difficult… hence another argument for a testbed
- We need strong version control on the reference dataset code
- General consensus is that meta-data associated with test code (description, classification etc..) should be separate (not embedded within) the source code

Reference Dataset (cont.)

- Metadata should include compiler options (for source code) , platform, and remediation info as a starting list of attributes
- Both “*In the Wild*” (open source) and “Manufactured” code should be part of the reference dataset
- Can we get necessary complexity in "manufactured code"? MIT Lincoln Lab’s research would indicate this may be possible
- Some complex code examples (not found “in the wild”) may have to be constructed "by hand"
- Would a CVS repository be enough? Likely not. Tagging test cases with additional information will be essential

Reference Dataset (cont.)

- **Collaboration on a reference dataset:**
  - We should have a community front end (interface for contributors), where peer evaluation decides if a submitted piece of code is a vulnerability.
  - Editorial control can be given to trusted users only. We need moderators.
  - The code will go through a review process, then after examination it can be accepted as a full part of the dataset.
  - Consensus should be used to determine changes to reference dataset
  - The security aspect must be considered for submitted code (virus in dataset code)
  - We will need a “caveat emptor” (buyer beware) for any testbed code (source or binary)

Reference Dataset (cont.)

- NIST has started development of a SAMATE Reference Dataset (SRD) Repository
  - Publicly available at http://samate.nist.gov
- Populating SRD with code samples from:
  - MIT Lincoln Lab (1000+ test cases)
  - Fortify Software (80+ test cases)
  - Klocwork Inc. (40 test cases)
  - Other possible sources
    - OWASP WebGoat
    - Foundstone HacmeBook and HacmeBank
    - CVE listing of known vulnerabilities
    - Other tool developer contributions

Defining Metrics for Software

- Code metric must be multi-dimensional; no single scalar metric can measure software assurance
- Ideally, confidence in tools should be high enough that Common Criteria EAL 2 should be a “tool-level application review”
- Some tool vendors do not use “pass/fail reporting” in their tools so flaws/vulnerabilities are simply listed
- Some code metrics are less controversial: tools can measure for an application’s effective “countermeasures” (e.g. canaries to check for buffer overflow), or “resiliency” as a metric.
- Hard to get agreement on "global metrics“ for software.
  - Metrics don’t have to be global (can be local to a reviewing community’s needs)

Defining Metrics for SwA Tools

- Tool run time vs. program complexity may be a poor measure of a tool's scalability, since some tools run very fast in practice against complex applications.
  - Perhaps just run-time/KLOC would be a better metric
- Other possible tool metrics include:
  - Workload (use of memory)
  - Completeness (what number of total flaws a tool detects)
  - Positive predictability (% true positives a tool detects)
  - Cost
  - Ease of use
  - Support
- SAMATE workshop members volunteer for souce code scanning tool metrics focus group

Next Steps

- Create draft functional specification for source code scanning tools
- Develop test plan and test suites for source code scanning tools
- Populate SAMATE SRD with test cases
- Identify gaps in today's tool capabilities
- Work toward common metrics for applications and SwA Tools

Participation

- Contact SAMATE project leader Paul Black (paul.black@nist.gov)
- Welcome SCADA community to participate in DHS SwA workshops and SAMATE project