VOLUME 5 NUMBER 3
OCTOBER 2017

CSIAC JOURNAL

A Quarterly Publication of the Cyber Security & Information Systems Information Analysis Center

DoD Software Assurance (SwA) Community of Practice:

TOOLS & TESTING TECHNIQUES

ASSURED SOFTWARE

VOLUME 2

CSIAC

Cyber Security & Information Systems Information Analysis Center

ABOUT THE CSIAC

As one of three DoD Information Analysis Centers (IACs), sponsored by the Defense Technical Information Center (DTIC), CSIAC is the Center of Excellence in Cyber Security and Information Systems. CSIAC fulfills the Scientific and Technical Information (STI) needs of the Research and Development (R&D) and acquisition communities. This is accomplished by providing access to the vast knowledge repositories of existing STI as well as conducting novel core analysis tasks (CATs) to address current, customer focused technological shortfalls.

OUR MISSION

CSIAC is chartered to leverage the best practices and expertise from government, industry, and academia in order to promote technology domain awareness and solve the most critically challenging scientific and technical (S&T) problems in the following areas:

- Cybersecurity and Information Assurance
- Software Engineering
- Modeling and Simulation
- Knowledge Management/Information Sharing

The primary activities focus on the collection, analysis, synthesis, processing, production and dissemination of Scientific and Technical Information (STI).

OUR VISION

The goal of CSIAC is to facilitate the advancement of technological innovations and developments. This is achieved by conducting gap analyses and proactively performing research efforts to fill the voids in the knowledge bases that are vital to our nation. CSIAC provides access to a wealth of STI along with expert guidance in order to improve our strategic capabilities.

WHAT WE OFFER

We provide expert technical advice and assistance to our user community. CSIAC is a competitively procured, single award contract. The CSIAC contract vehicle has Indefinite Delivery/Indefinite Quantity (ID/IQ) provisions that allow us to rapidly respond to our users' most important needs and requirements.

Custom solutions are delivered by executing user defined and funded CAT projects.

CORE SERVICES

- Technical Inquiries: up to 4 hours free
- Extended Inquiries: 5 - 24 hours
- Search and Summary Inquiries
- STI Searches of DTIC and other repositories
- Workshops and Training Classes
- Subject Matter Expert (SME) Registry and Referrals
- Risk Management Framework (RMF) Assessment & Authorization (A&A) Assistance and Training
- Community of Interest (COI) and Practice Support
- Document Hosting and Blog Spaces
- Agile & Responsive Solutions to emerging trends/threats

PRODUCTS

- State-of-the-Art Reports (SOARs)
- Technical Journals (Quarterly)
- Cybersecurity Digest (Semimonthly)
- RMF A&A Information
- Critical Reviews and Technology Assessments (CR/TAs)
- Analytical Tools and Techniques
- Webinars & Podcasts
- Handbooks and Data Books
- DoD Cybersecurity Policy Chart

CORE ANALYSIS TASKS (CATS)

- Customer tailored R&D efforts performed to solve specific user defined problems
- Funded Studies - $1M ceiling
- Duration - 12 month maximum
- Lead time - on contract within as few as 6-8 weeks

CONTACT INFORMATION

266 Genesee Street
Utica, NY 13502

1 (800) 214-7921

info@csiac.org

/DoD_CSIAC

/CSIAC

/CSIAC

DoDIACs
Department of Defense Information Analysis Centers

ABOUT THE JOURNAL OF CYBER SECURITY AND INFORMATION SYSTEMS

ABOUT THIS PUBLICATION

The Journal of Cyber Security and Information Systems is published quarterly by the Cyber Security and Information Systems Information Analysis Center (CSIAC). The CSIAC is a Department of Defense (DoD) Information Analysis Center (IAC) sponsored by the Defense Technical Information Center (DTIC) and operated by Quanterion Solutions Incorporated in Utica, NY.

ARTICLE REPRODUCTION

Cyber Security and Information Systems
266 Genesee Street
Utica, NY 13502

Phone: 800-214-7921
Fax: 315-732-3261
E-mail: info@csiac.org

An archive of past newsletters is available at **https://www.csiac.org/journal/.**

*To unsubscribe from CSIAC Journal Mailings please email us at **info@csiac.org** and request that your address be removed from our distribution mailing database.*

Journal of Cyber Security and Information Systems

Design and Development Process for Assured Software - Volume 11

INTRODUCTION

DESIGN AND DEVELOPMENT PROCESS FOR ASSURED SOFTWARE - VOLUME 2

By: Robert Gold,
Director, Engineering Enterprise in the Office of the Deputy Assistant Secretary of Defense for Systems Engineering, ODASD(SE)

Greetings,

It is my honor to introduce the second of two special software assurance (SwA) editions of the Journal of Cyber Security & Information Systems, published by the Cyber Security & Information Systems Information Analysis Center (CSIAC).

Our systems continue to increase their reliance on software – software was 66% of total system cost in 2010, software is projected to be 88% of system cost by 2024. Simultaneously, Department of Defense (DoD) systems have become progressively more networked, and dependent on a complicated global supply chain. Securing software through assurance tools, methods, and practices has correspondingly become increasingly necessary to ensure we field systems free from vulnerabilities and malware. To make assurance an integral part of DoD software development, the DoD has established program protection and system security engineering (SSE) as key disciplines to assure technology, components, and information against compromise and exfiltration. SSE is, in part, accomplished through the cost-effective application of protection measures to mitigate risks from vulnerabilities and attacks. The mission of software assurance, in support of SSE, is to remediate all detectable vulnerabilities, defects, and weaknesses as early in program system engineering as technically feasible, for critical functions and components.

DoD acquisition Program Managers (PMs) and their staff are at the front lines for implementing these assurance measures throughout acquisition, sustainment, and operation. For example, PMs will implement the use of automated software vulnerability detection and analysis tools and ensure risk-based remediation of software vulnerabilities is planned and resourced in Program Protection Plans, included in contract requirements, and verified through iterative assessments. We provide guidance for programs describing how to tailor software assurance to requirements according to characteristics of their developmental systems.

Assurance capabilities brought together by the Joint Federated Assurance Center (JFAC) support the planning, contracting, operation, measurement, and reporting of SwA work, including what a program must provide in its Program Protection Plan. Through JFAC, world-class engineering and acquisition professionals are working together in a broad range of initiatives that develop and implement best-practices to help programs engineer-in software assurance from the earliest activities in acquisition. These JFAC initiatives continue to develop and update artifacts, methodologies, guidance, contracting language, visibility, metrics, assessments, S&T focus, and initiatives that engineer assurance into SE activities across the life cycle. JFAC is planning further results that will provide automated SwA tool to enhance those efforts.

The JFAC SwA Technical Working Group has been meeting at least biweekly for about 3 years and includes participation beyond the stakeholder Services and agencies to include other parts of Government. For DoD programs, this group helped develop the JFAC Charter, JFAC Congressional Report, the JFAC Concept of Operations, standardized operating procedures for assurance provider and program relationship, metrics for use by programs to show progress implementing SwA, and more. Recently, the Group published the DoD SwA Capability Gap Analysis that applies Service-wide and that brings the Services and several agencies together on definitions, language, operation, and thinking for how best to implement future innovation in assurance tools and technology for the benefit of programs. Next initiatives include the first DoD-wide SwA Guidebook and the JFAC Outreach Plan.

My personal thanks to the authors and teams who contributed to these SwA special editions and to the CSIAC for working with JFAC to make it possible. I hope you find the articles informative and useful, and that you will take advantage of the critical thought, methodology development, and practical improvements we have made in software assurance. We would like your feedback. If you have comments or questions please contact the DASD(SE) JFAC team at osd.atl.asd-re.se@mail.mil. Or, why not go to the JFAC website and write us a ticket! ■

SARD: THOUSANDS OF REFERENCE PROGRAMS FOR SOFTWARE ASSURANCE

By: Paul E. Black, National Institute of Standards and Technology, Gaithersburg, MD

One way to understand the strengths and limitations of software assurance tools is to use a corpus of programs with known bugs. The software developer can run a candidate tool on programs in the corpus to get an idea of the kinds of bugs that the tool finds (and does not find) and the false positive rate. The Software Assurance Reference Dataset (SARD) [16] at the National Institute of Standards and Technology (NIST) is a public repository of hundreds of thousands of programs with known bugs. This article describes the content of SARD, how to find specific material, and ways to use it.

Certain trade names and company products are mentioned in the text or identified. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology (NIST), nor does it imply that the products are necessarily the best available for the purpose.

SARD has over 170,000 programs in C, C++, Java, PHP, and C# covering more than 150 classes of weaknesses. Most of the test cases are synthetic programs of a page or two of code, but there are over 7,000 full size applications derived from a dozen base applications. Although not every vulnerability is indicated in every program, the vast majority of weaknesses are noted in metadata, which can be processed automatically. Users can search for test cases by language, weakness type, and many other criteria and can then browse, select, and download them.

The term "bug" is ambiguous. "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. A vulnerability is the result of one or more ***weaknesses*** in requirements, design, implementation, or operation." [14, page 4] In isolation, a piece of code may have a buffer overflow or command injection weakness, but because the input is filtered or only comes from a trusted source, it may not constitute a vulnerability, which is exploitable. In fact, it may be difficult to determine if a particular piece of code may be reachable at all. It may, in practice, even be dead code. Hence, we usually talk about weaknesses and leave larger, system level concerns for another discussion.

We first explain the goals and organization of SARD, then describe the very diverse content. After that, we give advice on how to find and use SARD cases, related work and collections, and future plans for SARD.

A vulnerability is a property of system security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited

SARD Philosophy and Organization

The SARD consists of test cases, which are individual programs. Each test case has "metadata" to label and describe it. Many test cases are organized into test suites. Some test cases share common files with other cases.

The code is typical quality. It is not necessarily pristine or exemplary, nor is it horrible. SARD is not a compiler test. For now, we ignore the question of language version, e.g., C99 vs. C11.

Users can search for test cases by programming language, weakness type, size, and several other criteria and can then browse, select, and download them. Users can access test suites, which are collections of test cases. We explain more in the section explaining how to use SARD content.

Many synthetic programs represent thousands of variations for different weakness classes. In theory only the code pertaining to the weakness need be examined to determine that it is, indeed, a weakness. However, analysis tools must handle an unbounded amount of surrounding code to find sources of sinks, determine conditions when the piece of code may be executed, etc. Many sets of synthetic programs have the same base weakness wrapped in different ***code complexities***. For instance, an uninitialized variable may be declared in one function and a reference passed to another function, where it is used. Other code complexities are when the weakness is wrapped in various types of loops or conditionals or uses different data types.

Test cases are labeled "good," "bad," or "mixed." A "bad" test case contains one or more specific weaknesses. A "good" test case is associated with bad cases, but the weaknesses are fixed. Good cases can be used to check false positives. A "mixed" test case has both, for instance, code with a weakness and the same code with the weakness fixed. Weaknesses are classified using the Common Weakness Enumeration (CWE) [7] ID and name. We plan to also list their Bugs Framework (BF) [3] class. Fig. 1 shows the result of searching for test cases. Clicking on 199265 displays that case, as shown in Fig. 2.

SARD is archival. That is, once a test case is deposited, it will not be removed or changed. That way, if research uses a test case, later researchers can access that exact test case, byte for byte, to replicate the results. This is important to determine if, say, a new technique is more powerful than a previous technique.

If problems are later found with a test case, a new version may be added. For instance, if an extraneous error is found in a test case or the test case uses an obsolete language feature, an alternate may be added that corrects the problem. The original can still be accessed, but its status is *deprecated*.

A test case is deprecated if it should not be used for new work. If a test case does not yet meet our documentation, correctness, and quality requirements, its status is ***candidate***. When it does, its status is ***accepted***. A user can expect that the documentation of an accepted test case contains:

- A description of the purpose of the test case.
- An indication that it is good (false alarm), bad (true positive), or mixed.
- Links to any associated test cases, e.g. the other half of a bad/good test case pair.
- The source code language, e.g. C, Java, or PHP.
- Instructions to compile, analyze, or execute the test, if needed. This may include compiler name/version, compiler directives, environment variable definitions, execution instructions, or other test context information.
- The weakness(es) class(es).
- If this is a "bad" or "mixed" test, the location of known weaknesses, e.g. file name and line number.

Source code for an accepted test case will:

- Compile (for compilable languages).
- Run without fatal error, other than those expected for an incomplete program.
- Not generate any warnings, unless the warnings are expected as part of the test.
- Contain the documented weakness if the test case is a bad or mixed test case.
- Contain no weaknesses at all if it is a good case.

Test Case ID	Submission Date	Language	Type of Artifact	Status	Description	Weakness	Bad / Good / Mixed
148725	2013-05-22	Java	Source Code	C	CWE: 90 LDAP Injection. BadSource: getParameter_Servlet Read ...	CWE-090: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	
148762	2013-05-22	Java	Source Code	C	CWE: 90 LDAP Injection. BadSource: getQueryString_Servlet Parse ...	CWE-090: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	
148799	2013-05-22	Java	Source Code	C	CWE: 90 LDAP Injection. BadSource: listen_tcp Read data using a ...	CWE-090: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	
148820	2014-08-01	C	Source Code	C	CVE-2010-1772	CWE-416: Use After Free	
148936	2014-08-01	C	Source Code	C	CVE-2013-1572	CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')	
149039	2014-08-01	PHP	Source Code	C	CVE-2013-7233	CWE-352: Cross Site Request Forgery	
199265	2016-07-22	C	Source Code	C	Defect Type: Pointer related defects. Defect Sub-type: Bad cast of af	CWE-401: Improper Release of Memory Before Removing Last Reference ('Memory Leak') CWE-465: Pointer Issues CWE-476: NULL Pointer Dereference CWE-561: Dead Code CWE-704: Incorrect Type Conversion or Cast	
199266	2016-07-22	C	Source Code	C	Defect Type: Pointer related defects. Defect Sub-type: Bad cast of af	CWE-401: Improper Release of Memory Before Removing Last Reference ('Memory Leak') CWE-465: Pointer Issues CWE-476: NULL Pointer Dereference CWE-561: Dead Code CWE-704: Incorrect Type Conversion or Cast	
199267	2016-07-22	C	Source Code	C	Defect Type: Inappropriate code. Defect Sub-type: Return value ...	CWE-252: Unchecked Return Value CWE-561: Dead Code CWE-562: Return of Stack Variable Address CWE-672: Operation on a Resource after Expiration or Release	
199268	2016-07-22	C	Source Code	C	Defect Type: Inappropriate code. Defect Sub-type: Return value ...	CWE-252: Unchecked Return Value CWE-561: Dead Code CWE-562: Return of Stack Variable Address CWE-672: Operation on a Resource after Expiration or Release	
199275	2016-07-22	C	Source Code	C	Defect Type: Resource management defects. Defect Sub-type: ...	CWE-416: Use After Free CWE-476: NULL Pointer Dereference CWE-561: Dead Code CWE-824: Access of Uninitialized Pointer	

Figure 1. A screen shot of test cases found for a search. It shows that cases have "metadata" or information such as, test case ID, source code language, status, description, weaknesses included, and an indication of whether it has weaknesses (bad), no weaknesses (good), or mixed.

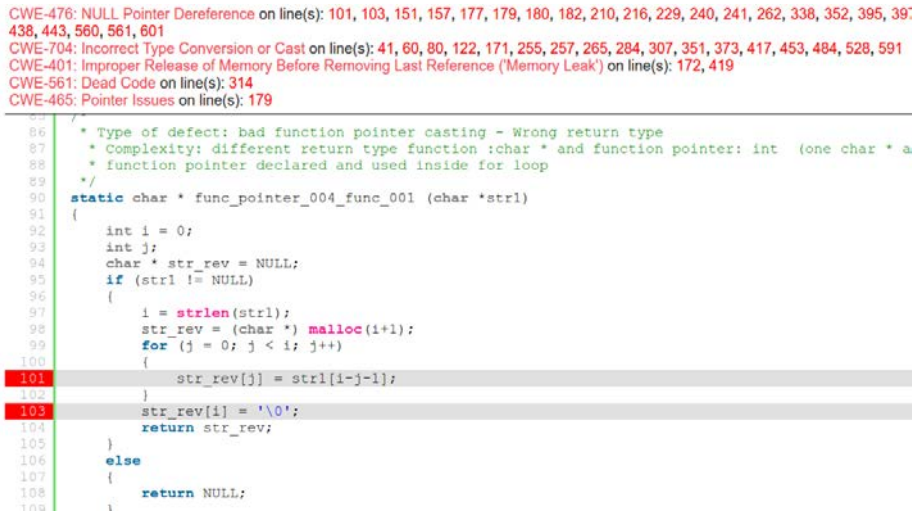


Figure 2. Screen shot of code from case 199625. The NULL pointer dereference weakness shows up on lines 101, 103, and other lines. Each case has such "metadata" available for for automatic processing.

We have permission to publicly furnish the SARD test cases. In fact, many test cases are in the public domain. We are working to attach explicit usage rights to each case and each suite.

SARD was designed to support almost one billion test cases. Cases are organized in directories of one thousand. For instance, when downloaded, the path to case 1320984 is `testcases/001/320/984`. That subdirectory may contain a single file, or it may contain many files and subdirectories for a large, complex case.

SARD Content

Since it is not clear what the perfect test suite would be (or if there is one!), we gathered many different test case collections from many sources. This section describes the collections to provide an idea of the kinds of cases that are currently available. First we describe the large collections of synthetic cases generated by programs. Next we describe the collections of cases written by hand. Finally we describe cases from production code. Table 1 gives a very general idea of all SARD cases listing the number of cases in each language. (The counts in the table do not include deprecated cases.) Fig. 3 gives a better idea of the quantity, size, and source of cases in each language.

Table 1. Number of SARD test cases in each programming language as of 30 July 2017.

Language	Number of Cases
C	46,846
C++	21,138
Java	28,828
PHP	42,248
C#	32,018

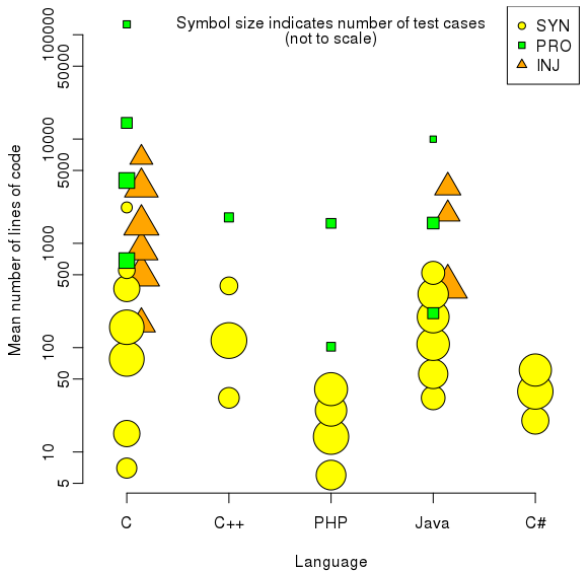


Figure 3. Number of test cases by language, clustered by lines of code. The Y axis is the mean lines of code of test cases in the cluster. Synthetic cases (SYN), in which all weaknesses are known, are yellow circles. Cases with weaknesses injected (INJ) into production code are orange triangles. Production code (PRO), which have some weaknesses identified, are green squares. The size of circles, triangles, and squares is the logarithm of number of test cases of that cluster; larger is more test cases.

By far the largest number of test cases are synthetic. One of our first collections came from MIT Lincoln Laboratory. They developed a taxonomy of code complexities and 291 basic C programs representing this taxonomy to investigate static analysis and dynamic detection methods for buffer overflows. Each program has four versions: a "good" version, accessing within bounds, and three "bad" versions, accessing just outside, moderately outside, and far outside the boundary of the buffer. These 1164 cases are explained in Kratkiewicz and Lippmann [11] and are designated test suite 89.

In 2011, the National Security Agency's Center for Assured Software (CAS) generated thousands of test cases in C/C++ and Java covering over 100 CWEs, called Juliet 1.0. (This was the tenth major SARD contribution and was named for the tenth letter of the International Radiotelephony Spelling Alphabet, which is "Juliet.") They can be compiled individually, in groups, or all together. Each case is one or two pages of code. They are grouped by language, then by CWE. In each CWE, base programs, using versions of printf() or different data types, are elaborated with up to 30 variants having complexities added. The following year they extended the collection with version 1.1, described in Boland and Black [4]. The latest version is Juliet 1.2, which comprises 61 387 C/C++ programs and 25 477 Java programs for almost two hundred weakness classes. They are test suites 86 (C/C++) and 87 (Java). The Juliet 1.0 and 1.2 suites are further described in documents at https://samate.nist.gov/SARD/around.php.

Following an architecture developed by NIST personnel and under their direction, a team of students at TELECOM Nancy, a computer engineering school of the Université de Lorraine, Nancy, France, implemented a generator that created many PHP cases. After that, other students rewrote the generator to be more modular and extensible, under guidance of members of the NIST Software Assurance Metrics And Tool Evaluation (SAMATE) team. They created a suite of 42 212 test cases in PHP covering the most common security weakness categories, including XSS, SQL injection, URL redirection, etc. These are suite 103 and are documented in Stivalet and Fong [20]. In 2016, SAMATE members oversaw additional work, again by TELECOM Nancy students, who created a suite of 32 003 cases in C#. These cases are suite 105.

Manually Written Cases

Many companies donated synthetic benchmarks that they developed manually. Fortify Software Inc., now HP Fortify, contributed a collection of C programs that manifest various software security flaws. They updated the collection as ABM 1.0.1. These 112 cases cover various software security flaws, along with associated "good" versions. These are test suite 6. In 2006, Klocwork Inc. shared 41 C and C++ cases from their regression suite. These are all a few lines of code to demonstrate use after free, memory leak, use of uninitialized variables, etc. Toyota InfoTechnology Center (ITC), U.S.A. Inc. created a benchmark in C and C++ for undefined behavior and concurrency weaknesses. The test suite, 104, has 100 test cases containing a total of 685 pairs of weaknesses. Each pair has a version of a function with a weakness and a fixed version of the function. For more details see [18]. The test cases are © 2012-2014 Toyota

InfoTechnology Center, U.S.A. Inc., distributed under the "BSD License," and added to SARD by permission. The SAMATE team noted coincidental weaknesses.

SARD also includes 329 cases from our static analyzer test suites [1]. These have suites for weaknesses, false positives, and weakness suppression in C (test suites 100 and 101), C++ (57, 58, and 59), and Java (63, 64, and 65).

SARD includes many small collections of synthetic test cases from various sources. Frédéric Michaud and Frédéric Painchaud, Defence R&D Canada, created and shared 25 C++ test cases. These test cases cover string and allocation problems, memory leaks, divide by zero, infinite loop, incorrect use of iterator, etc. These are test suite 62. Robert C. Seacord contributed 69 examples from "Secure Coding in C and C++" [17]. John Viega wrote "The CLASP Application Security Process" [22] as a training reference for the Comprehensive, Lightweight Application Security Process (CLASP) of Secure Software, Inc. SARD initially included 36 cases with examples of software vulnerabilities from use of hard-coded password and unchecked error condition to race conditions and buffer overflow. Many of the original cases have been improved and replaced. Hamda Hasan contributed 15 cases in C#, including ASP.NET, with XSS, SQL injection, command injection, and hard coded password weaknesses.

Cases From Production Software

All the cases described to this point were a few pages of code at most and were written specifically to serve as focused tests. Small synthetic cases may not show if a technique scales or if an algorithm can handle production code with complicated, interconnected data structures over thousands of files and variables. To fill this gap, SARD has cases that came from operational code.

MIT Lincoln Laboratory extracted 14 program slices from popular Internet applications (BIND, Sendmail, WU-FTP, etc.) with known, exploitable buffer overflows [23]. That is, they removed all but a relatively few functions, data structures, files, etc. so the remaining code ("the slice") has the overflow. They also made "good" (patched) versions of each slice. These 28 test cases are in SARD as test suite 88.

The Intelligence Advanced Research Projects Activity (IARPA) Securely Taking On New Executable Software Of Uncertain Provenance (STONESOUP) program created test suites in three phases. The goal of STONESOUP was to fuse static analysis, dynamic analysis, execution monitoring, and other techniques to achieve orders of magnitude greater assurance. For Phase 1 they developed five collections of small C and Java programs covering five vulnerabilities: memory corruption and null pointer dereference for C, and injection, numeric handling, and tainted data for Java. Each collection may be downloaded from the SARD Test Suites page and includes directions on how to compile and execute them and inputs that trigger the vulnerability. The test cases for Phase 2 were not particularly different from the Phase 1 cases.

For Phase 3, STONESOUP injected thousands of weakness variants into 16 widely-used web applications, resulting in 3188 Java cases and 4582 C cases. The weaknesses covered 25 classes such as integer overflow, tainted data, command injection, buffer overflow, and null pointer. Each case is accompanied with inputs triggering the vulnerability, as well as "safe" inputs. Because the cases represent thousands of copies of full-sized applications, IARPA STONESOUP Phase 3 is distributed as a virtual machine with a complete testing environment: the base applications, all libraries needed to compile them, difference (delta) files with flaws, and a Test and Evaluation eXecution and Analysis System (TEXAS) to compile an executable from a difference file and the base app, binaries to monitor the execution, triggering and safe inputs, and expected outputs. This material, as well as results of STONESOUP, are described in documents available at https://samate.nist.gov/SARD/around.php.

The STONESOUP base applications are significant enough by themselves that we describe them here. They are available from the Test Suite page as Standalone apps. These 15 apps are GNU grep, OpenSSL, PostgreSQL, Tree (a directory listing command), wireshark, Coffee MUD (Multi-User Dungeon game), Elastic Search, Apache Subversion, Apache Jena, Apache JMeter, Apache Lucene, POI (Apache Java libraries for reading and writing files in Microsoft Office formats), FFmpeg (a program to record, convert, and stream audio and video), and Gimp (GNU Image Manipulation editor). Application ID 16, JTree, is different from the others. It is a smaller form of STONESOUP, that is, a single base case injected with weaknesses. The base case is a Java version of Tree. When processed with unzip, ID 16 produces 34 subdirectories, each with difference files to create a version of JTree with an injected weakness. Running the included generate_application_testcases.py creates different versions of JTree, along with test material.

For Static Analysis Tool Expositions (SATE) [14], SAMATE members tracked vulnerabilities reported through Common Vulnerability and Exposures (CVE) [6] to source code changes. This resulted in 228 CVEs in WordPress, Openfire, JSPWiki, Jetty, Apache Tomcat, Wireshark (1.2 and 1.8), Dovecot, Chrome, and Asterisk. Each of these programs has its own test suite. Each CVE has one test case that contains the file or files with the vulnerabilities. The first test case in each suite has all the CVEs, files, and identified vulnerabilities for that program. These 10 test suites represent hundreds of reported, known vulnerabilities and the corresponding source code.

How to Use SARD Test Cases and Test Suites

The first step is to decide what test case properties are important to your situation. Programming language is the most obvious characteristic. Clicking on the "Search" tab, you may search SARD by many criteria, such as programming language, type of weakness, words in the description, type (bad, good, or mixed), status, and test case IDs, as shown in Fig. 4. Test case IDs may be ranges or lists. The type of weakness is matched to CWE descriptions as you type. You can search for and select those weaknesses that are most crucial in your situation.

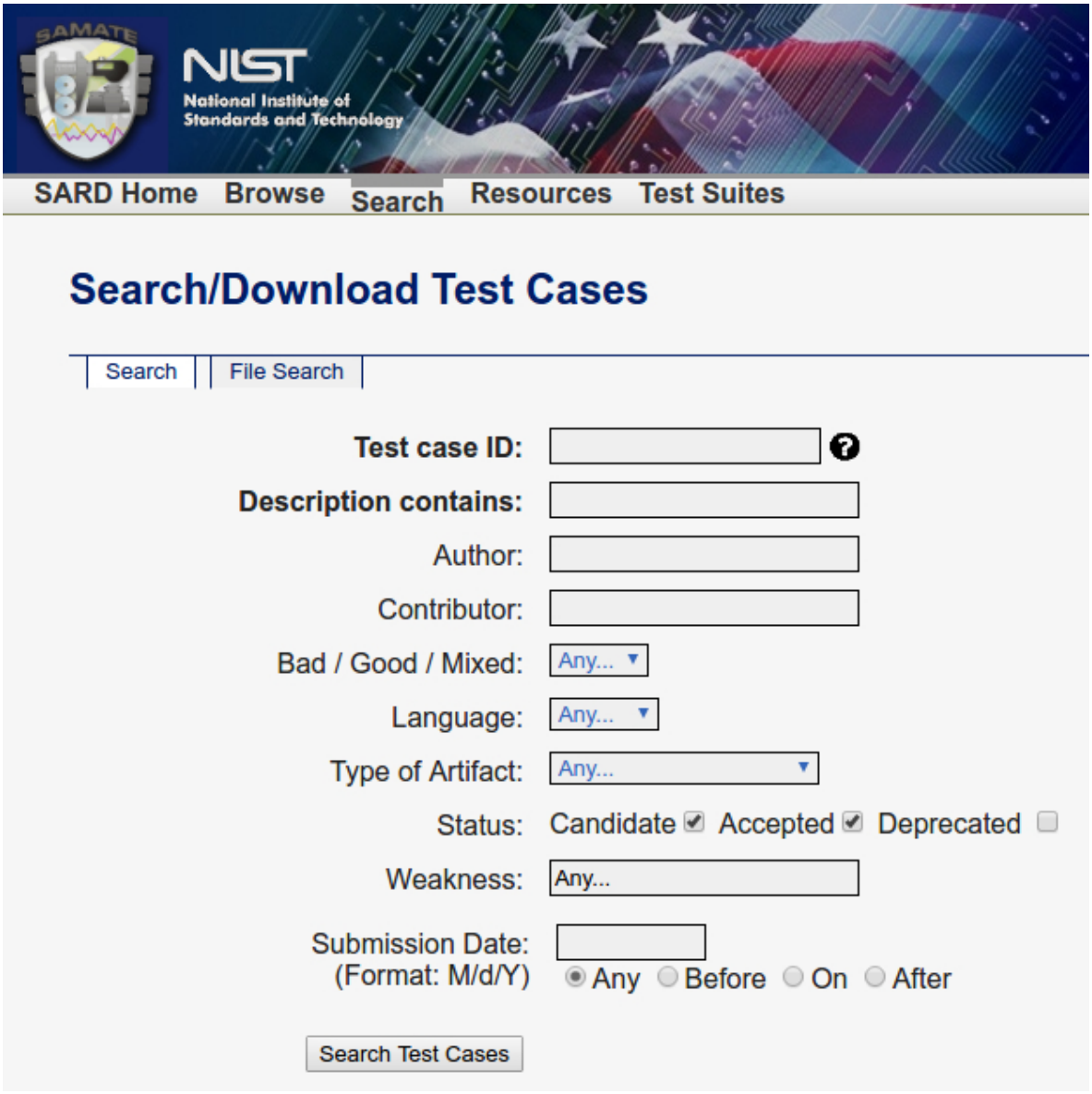


Figure 4. SARD search page. User may search for test cases meeting any combination of these criteria. Source: https://samate.nist.gov/SARD/index.php

Matching test cases are displayed as in Fig. 1. You may browse, select, and download any or all of the resulting cases. The download is a zip file containing a manifest (an XML listing of test cases and weakness locations) and the cases in a hierarchical directory structure of thousands described earlier.

In the File Search page, you can search for cases having files with certain names, sizes, or numbers of files, as shown in Fig. 5. This kind of search is useful if you are trying to find, say, very large test cases. We search by file name to find where test files come from or to find related cases, which often have files with similar names.

The SARD Test Suites page lists stand-alone suites, which are very large, test suites that are collections of test cases, and web and mobile applications. The web and mobile sections are for large (full-sized) applications that we will host in those domains. Standalone apps currently consists of STONESOUP base cases, described previously. The test suites page also has links to old collections that have been superseded.

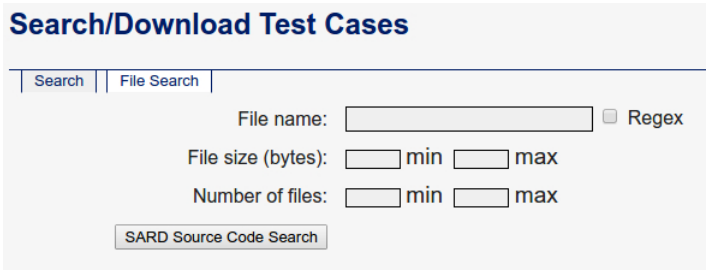


Figure 5. SARD file search page. User may search for test cases having files with particular names, files of particular sizes (minimum, maximum, or both), and particular numbers of files. User may give a regular expression to match file names, but a regex search is far slower. Source: https://samate.nist.gov/SARD/index.php

Paraphrasing Boland et. al. [4], many test suites, such as Juliet, are structured so that all the small test cases can be analyzed or compiled as a single, large program. This helps assess how a software-assurance

tool performs on larger programs. Because of the number of files and size of code, some tools might not be able to analyze all these test cases as a single program. Another use is to analyze separate test cases individually or in groups.

Because the manifest indicates where flaws occur, users can evaluate tool reports semiautomatically. When users run a source code analysis tool on a test case, the desired result is for the tool to report one or more flaws of the target type. A report of this type might be considered a true positive. If the tool doesn't report a flaw of the target type in a bad method, it might be considered a false negative. Ideally, the tool won't report flaws of the target type in a "good" test case or function; a report of this type might be considered a false positive. Because flawed and similar unflawed code might be in infeasible or "dead" code, users' policies on warnings about infeasible code must be taken into account.

As an illustration of using SARD, we offer our development of a small number of cases to show that a tool is effective at finding stack-based buffer overflows. First, we downloaded all buffer overflow test cases from SARD. To expedite analysis, we split every Juliet test case into two cases: one with only the bad code and one with only the good code. We also removed some unreachable code and conditional compilation commands. This resulted in 7338 test cases. We ran five tools on those cases. We compared, discussed, and grouped results until we came up with seven principles for selecting test cases [2]. This would have been far harder without the resources of SARD.

Related Work

We know of several other fixed collections of software assurance test cases. Some include tools to run experiments and compute results. After we itemize those collections, we list work to generate sets as needed.

The Software-artifact Infrastructure Repository (SIR) is "meant to support rigorous controlled experimentation with program analysis and software testing techniques, and education in controlled experimentation." [19] It provides Java, C, C++, and C# programs in multiple versions, along with testing tools, documentation, and other material. We found 85 objects, the most recent updated in 2015.

FaultBench "is a set of real, subject Java programs for comparison and evaluation of actionable alert identification techniques (AAITs) that supplement automated static analysis." [10] FaultBench has 780 bugs across six programs.

The OWASP Benchmark for Security Automation "is a free and open test suite designed to evaluate the speed, coverage, and accuracy of automated software vulnerability detection tools and services" [13]. It has 2740 small test cases, both with weaknesses and without weaknesses, in Java. It includes programs and scripts to run a tool and compute some results.

The Software Assurance Marketplace (SWAMP) provides more than 270 packages, in addition to the Juliet test suite, which is described above. "Packages are collections of files containing code to be assessed along with information about how to build the software package, if necessary." [21] There are packages in Java, Python, Ruby, C, C++, and web scripting languages. Each package may have multiple versions.

SARD approaches the problem of test cases by collecting a static set. An alternative is to generate sets of cases on demand. In theory, one could specify the language, weaknesses, code complexities, and other facets, and get a—potentially unique—set as needed. Generating sets on demand would be one way to address the concern that tool makers might add bits of code just to get a high score on a static benchmark. Generated cases could be automatically obfuscated, too. The disadvantage is that each generated set would have to be examined to be sure that the cases serve their purpose (or else the generator itself would have to be qualified, which is much harder). In practice, code generators or bug injectors are enormously difficult. Nevertheless, there is some work.

Large-Scale Automated Vulnerability Addition (LAVA) creates corpora by injecting large numbers of bugs into existing source code [9]. EvilCoder also injects bugs into existing code, although it injects bugs by locating guard or checking code and selectively disabling it [15].

The test generators implemented by TELECOM Nancy students are the source of large SARD test suites in PHP and C# [20]. The Department of Homeland Security's Static Tool Analysis Modernization Project (STAMP) recently awarded a contract to GrammaTech that includes development of a test case generator, Bug Injector [8]. KDM Analytics Inc. is enhancing their test case generator, TCG, for CAS. The latest version, TCG 3.2, produces both "bad" (flawed) and "good" (false positive) cases in C, C++, Java, and C# with control, data, and scope complexities [5]. TCG can generate millions of cases covering some three dozen Software Fault Pattern (SFP) clusters [12] and many CWEs for either or both Linux and Microsoft Windows platforms. Generated cases don't have a main() function, which allows cases to be compiled individually or as one large program.

Future of SARD

SARD began in 2006 in order to collect test cases for the NIST SAMATE. We had planned to collect artifacts from all phases of the software development life cycle, including designs, source code, and binaries, in order to evaluate assurance tools for all of those. We still leave that option open, but have not yet found many tools and pressing needs for other phases.

We plan to add cases in more languages, such as JavaScript, Ruby, Swift, Objective-C, Python, or Haskell. Which language

depends on the availability of test cases and the need. We are also adding thousands of mobile app test cases, since their architecture, implementation languages, and major threats are so different from typical applications.

We invite developers and researchers to donate their collections to SARD. It is a loss to the community when someone puts a lot of effort into developing a collection, then, after several years and project changes, the collection is lost.

Currently weaknesses are labeled by CWE. We will add labels of Bugs Framework (BF) classes [3] when it is more complete.

Conclusion

Analysts, users, and developers have cut months off the time needed to evaluate a tool or technique using test cases from SARD. SARD has been used by tool implementers, software testers, security analysts, and four SATEs to expand awareness of static analysis tools. Educators can refer students to SARD to find examples of weaknesses. Having a reliable and growing body of code with known weaknesses helps the community improve software assurance tools themselves and encourage their appropriate use. ■

Acknowledgements

We thank David Flater for making the chart in Fig. 3 and Charles de Oliveira for elucidating much of the SARD content.

REFERENCES

[1] Paul E. Black, Michael Kass, Hsiao-Ming (Michael) Koo, and Elizabeth Fong, "Source Code Security Analysis Tool Functional Specification Version 1.1," NIST Special Publication 500-268 v1.1, February 2011. DOI 10.6028/NIST.SP.500-268v1.1

[2] Paul E. Black, Hsiao-Ming (Michael) Koo, and Thomas Irish, "A Basic CWE-121 Buffer Overflow Effectiveness Test Suite," Proc. Sixth Latin-America Symposium on Dependable Computing (LADC 2013), April 2013.

[3] Irena Bojanova, Paul E. Black, Yaacov Yesha, and Yan Wu, "The Bugs Framework (BF): A Structured Approach to Express Bugs," August 2016, 2016 IEEE Int'l Conference on Software Quality, Reliability, and Security (QRS 2016), Vienna, Austria. DOI 10.1109/QRS.2016.29

[4] Tim Boland and Paul E. Black, "Juliet 1.1 C/C++ and Java Test Suite," October 2012, IEEE Computer, 45(10):88-90. DOI 10.1109/MC.2012.345

[5] Djenana Campara, private communication, 20 March 2017.

[6] "Common Vulnerabilities and Exposures: The Standard for Information Security Vulnerability Names," https://cve.mitre.org/, accessed 15 March 2017.

[7] "Common Weakness Enumeration: A Community-Developed List of Software Weakness Types," https://cwe.mitre.org/, accessed 10 March 2017.

[8] "DHS S&T Awards ITHACA, NY, Company $8M to Modernize Open-Source Software Static Analysis Tools," https://www.dhs.gov/science-and-technology/news/2017/03/07/st-awards-ithaca-ny-company-8m, accessed 15 March 2017.

[9] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan, "LAVA: Large-Scale Automated Vulnerability Addition," 2016 IEEE Symposium on Security and Privacy, San Jose, CA, 2016, pp. 110-121. DOI: 10.1109/SP.2016.15

[10] "FaultBench," http://www.realsearchgroup.org/faultbench/, accessed 16 March 2017.

[11] Kendra Kratkiewicz and Richard Lippmann, "A Taxonomy of Buffer Overflows for Evaluating Static and Dynamic Software Testing Tools," Proc. Workshop on Software Security Assurance Tools, Techniques, and Metrics, Elizabeth Fong, ed., NIST Special Publication 500-265, February 2006. DOI 10.6028/NIST.SP.500-265

[12] Nikolai Mansourov and Djenana Campara, "System Assurance: Beyond Detecting Vulnerabilities," Morgan Kaufmann, 2010, pp. 176–188.

[13] "OWASP Benchmark Project," https://www.owasp.org/index.php/Benchmark, accessed 16 March 2017.

[14] Vadim Okun, Aurelien Delaitre, and Paul E. Black, "Report on the Static Analysis Tool Exposition (SATE) IV," NIST Special Publication 500-297, January 2013. DOI 10.6028/NIST.SP.500-297

[15] Jannik Pewny and Thorsten Holz, "EvilCoder: Automated Bug Insertion," Proc. 32nd Annual Conference on Computer Security Applications (ACSAC '16), Los Angeles, CA, December, 2016, Pages 214-225. DOI: 10.1145/2991079.2991103

[16] "Software Assurance Reference Dataset," https://samate.nist.gov/SARD/, accessed 3 March 2017.

[17] Robert C. Seacord, "Secure Coding in C and C++," Addison-Wesley, 2005.

[18] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu, "Test Suites for Benchmarks of Static Analysis Tools," IEEE Int'l Symposium on Software Reliability Engineering (ISSRE '15), DOI: 10.1109/ISSREW.2015.7392027

[19] "Software-artifact Infrastructure Repository," http://sir.unl.edu/, accessed 16 March 2017.

[20] Bertrand Stivalet and Elizabeth Fong, "Large Scale Generation of Complex and Faulty PHP Test Cases," April 2016, 2016 IEEE Int'l Conference on Software Testing, Verification and Validation (ICST), Chicago, IL. DOI: 10.1109/ICST.2016.43

[21] "SWAMP Packages," https://www.mir-swamp.org/#packages/public, accessed 16 March 2017.

[22] John Viega, "The CLASP Application Security Process," Security Software, Inc., 2005.

[23] Misha Zitser, Richard Lippmann, and Tim Leek, "Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code," October/November 2004, Proc. SIGSOFT '04/12th Int'l Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12), Newport Beach, CA, pp 97-106. DOI 10.1145/1029894.1029911

IMPROVING SOFTWARE ASSURANCE THROUGH STATIC ANALYSIS TOOL EXPOSITIONS

By: Terry S. Cohen, Damien Cupif, Aurelien Delaitre, Charles D. De Oliveira, and Elizabeth Fong, Vadim Okun,
National Institute of Standards and Technology, Gaithersburg, MD

The National Institute of Standards and Technology Software Assurance Metrics and Tool Evaluation team conducts research in static analysis tools that find security-relevant weaknesses in source code. This article discusses our experiences with Static Analysis Tool Expositions (SATEs) and how we are using that experience to plan SATE VI. Specifically, we address challenges in the development of adequate test cases, the metrics to evaluate tool performance, and the interplay between the test cases and the metrics. SATE V used three types of test cases directed towards realism, statistical significance, and ground truth. SATE VI will use a different approach for producing test cases to get us closer to our goals.

DISCLAIMER: *Certain trade names and company products are mentioned in the text or identified. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology (NIST), nor does it imply that the products are necessarily the best available for the purpose.*

I. Introduction

Software assurance is a set of methods and processes to prevent, mitigate or remove vulnerabilities and ensure that the software functions as intended. Multiple techniques and tools should be used for software assurance [1]. One technique that has grown in acceptance is static analysis, which examines software for weaknesses without executing it [2]. The National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) project has organized five Static Analysis Tool Expositions (SATEs), designed to advance research in static analysis tools that find security-relevant weaknesses in source code. An analysis of SATE V in preparation of the upcoming SATE VI is reported here.

We first discuss our experiences with SATE V, including the selection of test cases, how to analyze the warnings from static analysis tools, and our results. Three selection criteria for the test cases were used: 1) code realism, 2) statistical significance, and 3) knowledge of the weakness locations in code (ground truth). SATE V used test cases satisfying any two out of the three criteria: 1) production test cases with real code and statistical significance, 2) CVE-selected test cases, with real code and ground truth, and 3) synthetic test cases with ground truth and statistical significance. We describe metrics that can be used for evaluating tool effectiveness. Metrics, such as precision, recall, discrimination, coverage and overlap, are discussed in the context of the three types of test cases.

Although our results from the different types of test cases in SATE V bring different perspectives on static analysis tool performance, this article shows that combining such perspectives does not adequately describe real-world use of such tools. Therefore, in SATE VI, we plan to produce test cases incorporating all three criteria, so the results will better reflect real-world use of tools. We discuss the approach we will use: injecting a large number of known, realistic vulnerabilities into real production software. Thus, we will have statistical significance, real code, and ground truth.

Background

Providing metrics and large amounts of test material to help address the need for static analysis tool evaluation is a goal of the National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) project's Static Analysis Tool Exposition (SATE). Starting in 2008, we have conducted five SATEs.

SATE, as well as this article, is focused on static analysis tools that find security-relevant weaknesses in source code. These weaknesses, unless avoided or removed early, could lead to security vulnerabilities in the executable software.

SATE is designed for sharing, rather than competing, to advance research in static analysis tools. Briefly, a team led by NIST researchers provides a test set to toolmakers, invites them to run their tools, and they return the tool outputs to us. We then perform partial analysis of tool outputs. Participating toolmakers and organizers share their experiences and observations at a workshop.

The first SATE used open source, production programs as test cases. We learned that not knowing the locations of weaknesses in the programs complicates the analysis task. Over the years, we added other types of test cases.

One type, CVE-selected test cases, is based on the Common Vulnerabilities and Exposures (CVE) [3], a database of publicly reported security vulnerabilities. The CVE-selected test cases are pairs of programs: an older *bad* version with publicly reported vulnerabilities (CVEs) and a *good* version, that is, a newer version where the CVEs were fixed. For the CVE-selected test cases, we focused on tool warnings that correspond to the CVEs.

A different approach is computer-assisted generation of test cases. In SATE IV and V, we used the Juliet test suite [4], which contains tens of thousands of synthetic test cases with precisely characterized

weaknesses. This makes tool warnings amenable to mechanical analysis. Like the CVE-selected test cases, there are both a bad version (code that should contain a weakness) and a good version (code that should not contain any weakness).

Initially, we had two language tracks: C/C++ and Java. We added the PHP track for SATE IV. In SATE V, we introduced the Ockham Criteria [5] to exhibit sound static analysis tools. Table 1 presents toolmaker participation over the years. The PHP track and the Ockham Criteria had one participant each in SATE V. Note, because SATE analyses grew in complexity and length, we changed from yearly SATEs (2008, 2009, and 2010) to the current nomenclature (IV, V, and VI).

Table 1: Number of tools participating per track over SATEs

	Total	C/C+	Java
2008	9	4	7
2009	8	5	5
2010	10	8	4
IV	8	7	3
V	14	11	6

Related Work

Software weaknesses can lead to vulnerabilities, which can be exploited by hackers. Definition and classification of security weaknesses in software is necessary to communicate and analyze tool findings. While many classifications have been proposed, Common Weakness Enumeration (CWE) is the most prominent effort [6, 7]. The Common Vulnerabilities and Exposures (CVE) database, comprised of publicly reported security vulnerabilities, was discussed in the Background section. While the CVE database includes specific vulnerabilities in production software, the CWE classification system lists software weakness types, providing a common nomenclature for describing the type and functionality of CVEs to the IT and security communities.

For example, CVE-2009-2559 is a buffer overflow vulnerability in Wireshark, which can be used by hackers to cause denial of service (DoS) [8]. CVE-2009-2559 is associated with two CWEs: CWE-126: Buffer Over-read [9], which is caused by CWE-834: Excessive Iteration [10]. The NIST National Vulnerability Database (NVD) described it using CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer [11, 12], which is a parent of CWE-126. We describe our use of CVEs and CWEs in our Methodology section.

Researchers have collected test suites and evaluated static analysis tools. Far from attempting a comprehensive review, we list some of the relevant studies here.

Kratkiewicz and Lippmann developed a comprehensive taxonomy of buffer overflows and created 291 test cases - small C programs - to evaluate tools for detecting buffer overflows [13]. Each test case has three vulnerable versions with buffer overflows just outside, moderately outside, and far outside the buffer, and a fourth, fixed, version. Their taxonomy lists different attributes, or *code complexities*, including aliasing, control flow, and loops, which may complicate analysis by the tools.

The largest synthetic test suite in the NIST Software Assurance Reference Dataset (SARD) [14] was created by the U.S. National Security Agency's (NSA) Center for Assured Software (CAS). Juliet 1.0 consists of about 60 000 synthetic test cases, covering 177 CWEs and a wide range of code complexities [4]. CAS ran nine tools on the test suite and found that static analysis tools differed significantly with respect to precision and recall. Also, tools' precision and recall ordering varied for different weaknesses. CAS concluded that sophisticated use of multiple tools would increase the rate of finding weaknesses and decrease the false positive rate. A newer version of the test suite, Juliet 1.2, correcting several errors and covering a wider range of CWEs and code constructs, was used in SATE V.

Rutar et al. ran five static analysis tools on five open source Java programs, including Apache Tomcat, of varying size and functionality [15]. Due to many tool warnings, they did not categorize every false positive and false negative reported by the tools. Instead, the tool outputs were cross-checked with each other. Additionally, a subset of warnings was examined manually. One of the conclusions by Rutar et al. was that there was little overlap among warnings from different tools. Another conclusion was that a meta-tool combining and cross-referencing output from multiple tools could be used to prioritize warnings [15].

Kupsch and Miller evaluated the effectiveness of static analysis tools by comparing their results with the results of an in-depth manual vulnerability assessment [16]. Of the vulnerabilities found by manual assessment, the tools found simple implementation bugs, but did not find any of the vulnerabilities requiring a deep understanding of the code or design.

Developing test cases is difficult. There have been many approaches. Zhen Li et al. developed VulPecker, an automated vulnerability detection system, based on code similarity analysis [17]. Their recent study focused on the creation of a Vulnerability Patch Database (VPD), comprised of over 1700 CVEs from nineteen C/C++ open source software. Their CVE-IDs are mapped to diff hunks, which are small files tracking the location of a given weakness and changes in source code across versions.

Instead of extracting CVEs from programs, some studies have looked at injecting vulnerabilities for static analysis tool studies. The Intelligence Advanced Research Projects Activity (IARPA) developed the Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) program [18] to inject realistic bugs into production software. The injected vulnerabilities were embedded in real control flow and data flow [19]. These seeded vulnerabilities were snippets of code showcasing a specific vulnerability. However, these embedded snippets were unrelated to

the original source program, limiting realism in injected weaknesses. These test cases can be downloaded from the SARD [14].

In preparation for SATE VI, the SATE team looked extensively at related approaches. One important project was from the MIT Lincoln Laboratory, which developed a large-scale automated vulnerability (LAVA) technique to automatically inject bugs into real programs [20]. The program uses a "taint analysis-based technique" to dynamically identify sites that can potentially hold a bug, and user-controlled data that can be used at those vulnerable locations to trigger the weakness. Thus, the triggering input and the vulnerability are both known. LAVA can inject thousands of bugs in minutes. However, the tool alters the program data flow and only supports a small subset of CWE classes related to buffer overflow, therefore, limiting the realism of the injected weaknesses.

The program uses a "taint analysis-based technique" to dynamically identify sites that can potentially hold a bug

Another automated bug insertion technique is EvilCoder, developed by the Horst Görtz Institut, Germany [21]. Using a static approach, EvilCoder computes code property graphs from C/C++ programs to create a graph database, containing information about types, control flows and data flows. The program identifies paths that could be vulnerable, but are currently safe. Bug insertion is accomplished by breaking or removing security checks, making a path insecure. The limitation of this static analysis-based approach is that it does not produce triggering inputs to demonstrate the injected bugs.

II Test cases

Tool users want to understand how effective tools are in finding weaknesses in source code. Based on our SATE experiences, a perfect test case satisfies three criteria.

First, for tool results to be generally applicable, test cases should be representative of real, existing software. In other words, they should be similar in complexity to real software.

Second, for tool results to be statistically significant, the test cases must contain many different weakness instances of various weakness types. Since CWE has hundreds of weakness classes and the weaknesses can occur in a wide variety of code constructs, large numbers of test cases are needed.

Finally, to recognize tools' blind spots, we need the ground truth – knowledge of all weakness locations in the software. In other words, without the ground truth we cannot know which weaknesses remain undetected by tools. Additionally, it greatly simplifies analysis of tool outputs by enabling mechanical matching, based on code locations and weakness types.

In summary, the three selection criteria for test cases are 1) realistic, existing code, 2) large amounts of test data to yield statistical significance, and 3) ground truth. Figure 1 illustrates these criteria. So far, we do not have test cases that satisfy all three criteria simultaneously. For SATE V, we have produced test cases satisfying any two out of the three criteria (Figure 1). We chose the following three types of test cases:

First, production software large enough for statistical significance and, by definition, representative of real software. However, the weaknesses in it are at best only partially known.

Second, a set of test cases (i.e., a test suite) mechanically generated, so that each test case contains one weakness instance embedded in a set of code complexities. We used the Juliet test suite, a diverse set of clearly identified weakness instances, for this set. This approach has ground truth and produces statistically significant results. However, the synthetic test cases may not be representative of real code.

Finally, CVE-selected test cases that contain vulnerabilities that were deemed important to be included in the CVE database. These test cases are real software and have ground truth. However, the determination of CVE locations in code is a time-consuming task, which makes it hard to achieve statistical significance.

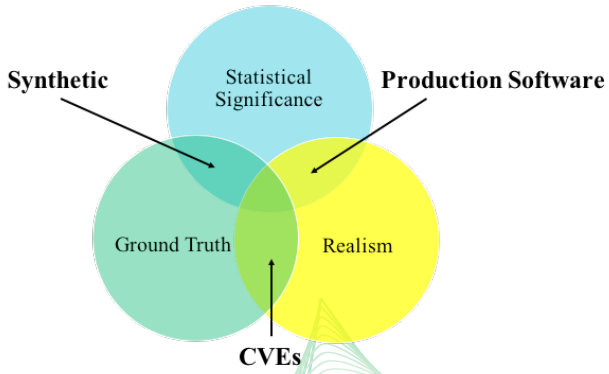


Figure 1: Types of test cases

III Metrics

To measure the value of static analysis tools, we need to define metrics to decide which attributes and characteristics should be considered. For SATE analyses, we established a universal way of measuring the tools' output objectively. The following metrics address several questions about tool performance.

First, what types of weaknesses can a tool find? *Coverage* is measured by the number of unique weakness types reported over the total number of weakness types included in the test set.

Second, what proportion of weaknesses can a tool find? *Recall* is calculated by dividing the number of correct findings (true positives) by the total number of weaknesses present in the test set, i.e., the sum of the number of true positives (TP) and the number of false negatives (FN). *Recall = TP / (TP + FN)* [5].

Third, what proportion of covered flaws can a tool find? *Applicable recall* (App.Recall) is recall reduced to the types of weaknesses a tool can find. It is calculated by dividing the number of true positives (TP) by the number of weaknesses in the test set, which are covered by a tool. In other words, a tool's performance is not penalized if it does not report weaknesses that it does not look for (App.FN). *App. Recall = TP / (TP + App.FN)*

Fourth, how much can I trust a tool? *Precision* is the proportion of correct warnings produced by a tool and is calculated by dividing the number of true positives by the total number of warnings. The total number of warnings is the sum of the number of true positives (TP) and the number of false positives (FP). *Precision = TP / (TP + FP)*

Fifth, how smart is a tool? Bad and good code often look similar. It is useful to determine whether the tools can differentiate between the two. Although precision captures that aspect of tool efficiency, it is relevant only when good sites are prevalent over bad sites. When there is parity in the number of good and bad sites, e.g., in some synthetic test suites, a tool could indiscriminately flag both good and bad test cases as having a weakness and still achieve a precision of 50 %. *Discrimination*, however, recognizes a true positive on a particular bad test case only if a tool did not report a false positive on the corresponding good test case. A tool that flags every test case as flawed would achieve a discrimination rate of 0 %.

Finally, can tool findings be confirmed by other tools? *Overlap* represents the proportion of weaknesses found by more than one tool. The use of independent tools would find more weaknesses (higher recall), whereas the use of similar tools would provide a better confidence in the common warnings' accuracy.

Table 2 summarizes the applicability of the metrics on the three types of test cases.

Table 2: Mapping metrics to test case types

	Production Software	Software w/ CVEs	Synthetic Test Cases
Coverage	Limited	Limited	Applicable
Recall	N/A	Applicable	Applicable
Precision	Applicable	N/A	Applicable
Discrimination	N/A	Limited	Applicable
Overlap	Applicable	Applicable	Applicable

Figure 1 summarizes the types of test cases. The mapping of their metrics is clearly delineated in Table 2. Production software has realism and statistical significance, but no ground truth. CVE-selected test cases have realism and ground truth, but no statistical significance. Synthetic test cases have statistical significance and ground truth, but no realism.

Precision and overlap can be calculated for production software test cases. However, due to the lack of ground truths, recall and discrimination cannot be determined, and only limited results for coverage can be obtained. In contrast, because the CVE-selected test cases are real software with ground truth, both recall and overlap can be calculated. However, because locating vulnerabilities is both difficult and time-consuming, precision cannot be determined, and limited results can be obtained for coverage and discrimination. Although these metrics are applicable to synthetic test cases (i.e., can be calculated), these cases may not generalize to real-world software.

IV Test Case Results

Methodology

This section focuses on SATE V test case results from the C/C++ track. For this track, we had selected two common open source software programs for the production software analyses: Asterisk version 10.2.0, an IP PBX platform[2], and Wireshark version 1.8.0, a network traffic analyzer. Asterisk comprises over 500,000 lines of code; Wireshark contains more than 2 million lines of code. These test cases can be downloaded from the NIST Software Assurance Reference Dataset (SARD) [14]. For the CVE-selected test cases, we also asked toolmakers to run their tools on later, fixed versions of these test cases, using Asterisk version 10.12.2 and Wireshark version 1.8.7. We used the NSA CAS Juliet test set for the synthetic test cases [4].

Different methods were used to evaluate tool warnings depending upon the type of test case. As we discussed in Section II, synthetic test cases contain precisely characterized weaknesses. Metadata includes the locations where vulnerabilities occur, good and bad blocks of code, and CWEs. Consequently, the analysis of all warnings generated by tools is possible. For each test case, we selected tool findings if its CWE matched the corresponding test case's CWE group.

As pointed out in Section II, finding the locations of CVEs in pairs of good and bad code was a time-consuming process. The metadata from production software is rich enough to demonstrate whether a tool found a CVE through automatic analysis. However, because CVEs were few in number and tools did not uniformly report vulnerabilities, we also conducted manual analyses. For each CVE, we selected the tool finding reported at the corresponding lines of code, only considering the finding if its CWE and the CVE's CWE belonged to the same CWE group. Once found, an expert would confirm whether the automated analysis was correct. In addition to extracting CVE test cases this way, our experts also manually checked the code for matches missed by the algorithm. Our experts would

rate the CVEs as having been precisely identified or coincidentally (indirectly) identified.

The analysis of production test cases was different. Analyses of tool warnings and reporting were often labor-intensive and required a high level of expertise. A simple binary true/false positive verdict on tool warnings did not provide adequate resolution to communicate the relationship of the warning to the underlying weakness [22]. Because of the large number of tool warnings and the lack of ground truth, we randomly selected warnings from each tool report, based on the weakness category and the security rating. After sampling 879 warnings and manually reviewing their correctness, we assigned each warning to a warning category. A security warning was related to an exploitable security vulnerability. A quality warning was not directly related to security, but it required a software developer's attention. An insignificant classification referred to a true warning, but insignificant claim. A false warning rating corresponded to a false positive, and an unknown rating was one whose correctness could not be determined.

Results

SATE is not a competition. To prevent endorsement of the participating toolmakers, we anonymized data. The results generated from Tools A through H are reported here.

Figure 2 shows the precision vs. discrimination tool results for the synthetic test cases. The precision results are similar across all tools, whereas discrimination results are not. This is because the number of buggy sites is similar to the number of safe sites, as is the case for synthetic and CVE-selected test cases. Thus, discrimination is a better metric to differentiate tools. Note that for real software, most sites are safe and only a small proportion of sites are buggy, so precision would be very low if a tool reports a warning for every site, flawed or not.

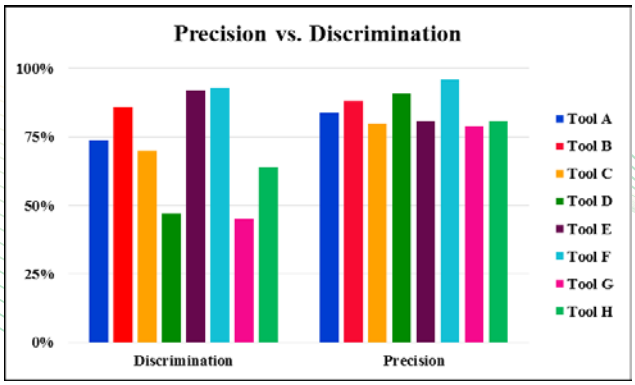


Figure 2: Precision vs. discrimination tool results for the Synthetic test cases - Source: Author(s)

The synthetic test cases offer an excellent demonstration of tool efficiency. Table 3 combines metric results from testing of the Juliet synthetic test suite. Tool F demonstrated the highest applicable recall and discrimination, but displayed the lowest coverage. Tool B, on the other hand, exhibited the broadest coverage and lower discrimination than that of Tool F.

Table 3: Applicable recall, coverage, and discrimination for the Synthetic test cases - Source: Author(s)

Tool	App. Recall	Coverage	Discrimination
Tool A	21%	29%	74%
Tool B	25%	42%	86%
Tool C	18%	22%	70%
Tool D	8%	19%	47%
Tool E	19%	15%	92%
Tool F	56%	9%	93%
Tool G	2%	35%	45%
Tool H	25%	31%	64%

Overlap identifies similar and independent tools. For example, if a vulnerability is reported by three tools, it is under "3 tools". There is an overlap when more than one tool correctly reports a weakness. Theoretically, the use of independent tools would find more weakness (higher recall), whereas the use of similar tools would provide better precision. Figure 3 shows the overlap distribution for synthetic test cases. Clearly, tools do not report the same weaknesses.

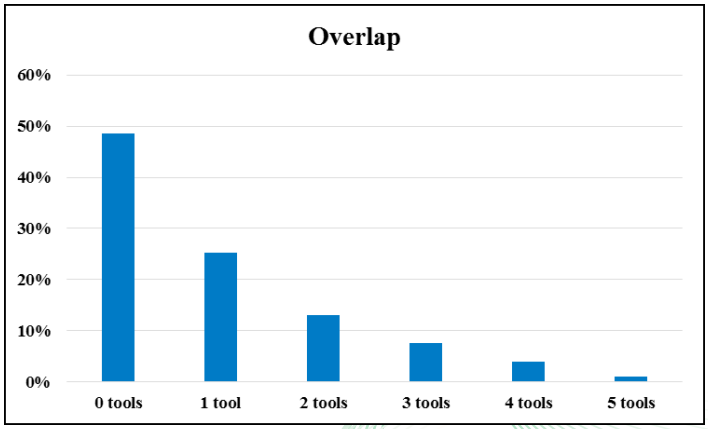


Figure 3: Overlap distribution for Synthetic test cases - Source: Author(s)

Because CWEs are broad in range and number, the SATE team grouped CWEs to analyze the SATE V results. Table 4 displays the nine CWE groups most represented in the CVE and synthetic test cases. Buffer operations, input validation, and numeric errors dominated the results. It should be noted that some of the weakness types under the loop and recursion CWE group could be very easy to detect, while others may be very difficult to detect, so results for these synthetic cases are lower than for other CWE groups.

Table 4: CWE Groups most represented in the CVE and Synthetic test cases - Source: Author(s)

CWE Group	CVE Count	Synthetic Count
Loop and recursion	42	488
Post buffer operation	39	13170
Numeric errors	27	7992
Ante buffer operation	21	4276
Input validation	11	9216
Invalid pointer	8	1406
Type-related	8	1384
Initialization	6	1141
Memory allocation	6	960

Figures 4 to 6 display the results for two metrics: recall and precision. The figures on the left provide a comparison of synthetic and CVE-selected test cases. The figures on the right provide a comparison of synthetic and production test cases. As examples, we use Tools B, H, and A to demonstrate the discrepancies between the results on different types of test cases. Recall was generally higher on synthetic test cases than in the CVE-related test cases. However, Tool A performed better with respect to CVEs in this case. Similarly, a comparison of the precision results indicates that the tools generated fewer false positives on the synthetic test cases than on the production test cases, leading to higher precision. Lower code complexity may account for the better recall and precision on the synthetic test cases compared to the CVE-related and production test cases.

Recall was generally higher on synthetic test cases than in the CVE-selected test cases. However, Tool A performed better with respect to CVEs in this case. Similarly, a comparison of the precision results indicates that the tools generated fewer false positives on the synthetic test cases than on the production test cases, leading to higher precision. Lower code complexity may account for the better recall and precision on the synthetic test cases compared to the CVE-selected and production test cases.

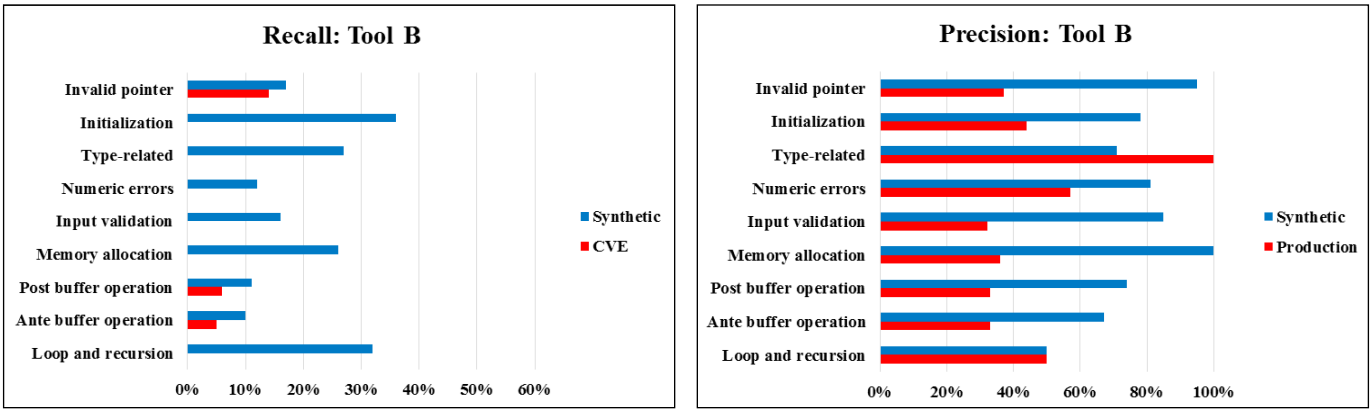


Figure 4: Recall for Synthetic vs. CVE test cases and precision for Synthetic vs. Production test cases - Source: Author(s)

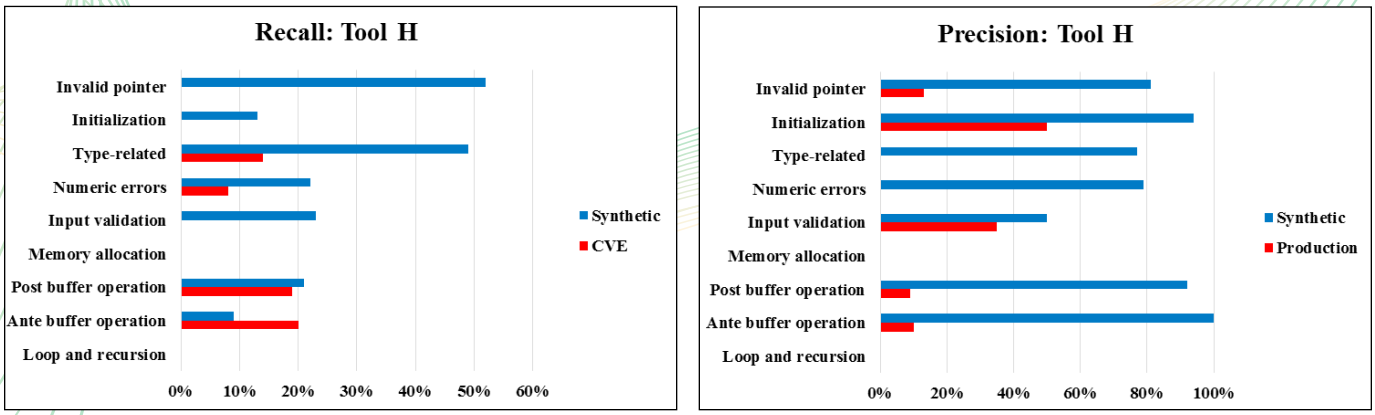


Figure 5: Recall for Synthetic vs. CVE test cases and precision for Synthetic vs. Production test cases - Source: Author(s)

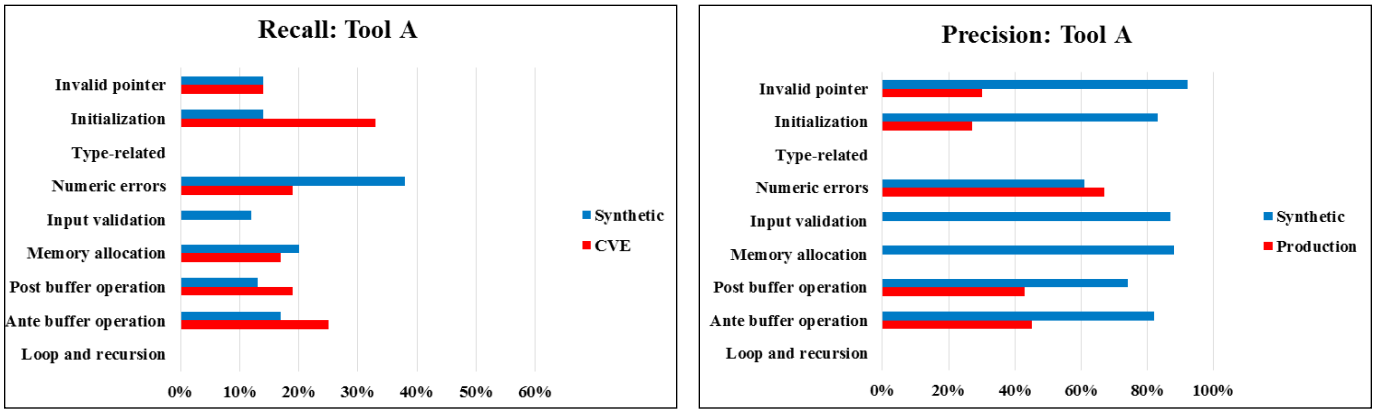


Figure 6: Recall for Synthetic vs. CVE test cases and precision for Synthetic vs. Production test cases - Source: Author(s)

Our examples illustrate the differences between the three types of test cases, making generalization challenging. For the production test cases, there was no ground truth, so tool recall could not be determined. Tools mostly reported different defects, so there was low overlap. Also, the results from synthetic cases may not generalize to real-world software. Clearly, characterizing a large set of CVE-selected test cases is very time consuming, so there was not enough test data collected for statistical significance. We will discuss a different approach in the context of our next SATE, SATE

VI Future SATE VI Plans

The lack of vulnerability corpora has always hampered researchers' work in software assurance, because high quality test data is essential to achieve meaningful studies applicable to real-world software development. The real challenge does not solely lie in having test cases at our disposal, but rather to have them display specific criteria: ground truth, bug realism, and statistical significance.

Our main goal for SATE VI is to improve the quality of our test suites by producing test cases satisfying these three criteria. Time is a critical factor in the development or selection of new test cases, their use by toolmakers, and the subsequent analysis and reporting of results. CVE extraction yields real bugs, however there are too few CVEs to showcase numerous bugs in a single version of software. Having to run tools on multiple versions of large test cases is time consuming and can be problematic for SATE.

Manual bug injection enables a greater number and diversity in real bugs, but also takes time and effort. To prepare test cases for SATE VI, our team is using a semi-automated process. For each class of weaknesses that we want to insert, the first step is to automatically identify sites that are currently safe, but could become vulnerable with manual transformation, as in EvilCoder [21]. A site is a conceptual place in a program where an operation is performed and a weakness might occur. For example, for C programs, every buffer access is a site where a buffer overflow might occur.

It is essential to understand that finding safe sites is much easier than finding vulnerable sites

The next step is to find execution paths leading to those sites. We will use guided fuzzing techniques to produce user inputs. Then, we will perform manual source code transformations, where the injected (or seeded) vulnerabilities will use the data flow and control flow of the original program. Finally, we will implement triggering and regression tests to demonstrate the injected bugs and check for conflicts between different injected bugs.

It is essential to understand that finding safe sites is much easier than finding vulnerable sites. Missing a safe site only represents the loss of one potential injected bug. To identify those sites, we must analyze our program the way a compiler does. To achieve this, we are analyzing the abstract syntax tree (AST) and extracting specific patterns. Ultimately, we want to use those sites to guide manual bug injection.

Identifying a site does not provide the input leading to it. We plan to use fuzzing tools to determine such input.

Our team will gather a set of CVEs and extract real-world insecure patterns to mimic production software vulnerabilities. Source transformations will be performed manually to reproduce common industry practices and yield realistic injected bugs. To achieve this, we will verify that the seeded vulnerabilities do not significantly alter the original data flow and control flow of the target program.

We must demonstrate that a given input leads to a real vulnerability. Manual bug injection requires much effort and high-level analysis to produce exploits. In fact, demonstrating exploitability is very challenging for static analyzers. Therefore, it is sufficient to demonstrate that our program exhibits abnormal behavior due to injected bugs. Consider this: an off-by-one buffer overflow will not always result in a program crashing, however, it can be validated using an assert statement.

VI Conclusion

In this article, we have discussed our experiences with SATE that can be useful for the software assurance community. Specifically, the article focused on the selection of test cases and how to analyze the output warnings from tools. We described metrics that could be used for evaluating tool effectiveness. Because tools report different weaknesses, there is little overlap in results.

SATE V covered three types of test cases: 1) production test cases, which had real code and statistical significance, 2) CVE-selected test cases, which had real code and ground truth, and 3) synthetic test cases, which had both ground truth and statistical significance. Although synthetic test cases cover a broad range of weaknesses, such test cases cannot be generalized to real-world software, like production cases. CVE extraction yields real bugs in production software, but it is both time-consuming and generates no statistical significance. Finally, static analysis tools can identify a large number of warnings in production software, which is real code. However, we do not know the location of all vulnerabilities, i.e., ground truth. Therefore, we require a better test suite, covering all three criteria for test cases.

Our main goal for future SATEs is to improve the quality of our analyses by producing test cases satisfying all three criteria. We believe inserting security-relevant vulnerabilities into real-world software can help us achieve this goal.

We learned through the study of three sophisticated and fully-automated injection techniques that the injected bugs are either insufficiently realistic [18, 20] or lack triggering inputs [21]. Purely manual injection has the benefit of yielding more realistic bugs, however it is time-consuming. Our team is considering a semi-automated process, speeding the discovery of potential sites, so we can perform manual source code transformations. In particular, we want to make sure that the seeded vulnerabilities do not significantly alter the data flow and control flow of the original program, and programming follows common development practices. Since demonstrating the injected bugs is essential, we will ensure that the injected bugs trigger abnormal program behavior.

REFERENCES

[1] Larsen, G., Fong, E. K. H., Wheeler, D. A., & Moorthy, R. S. (2014, July). State-of-the-art resources (SOAR) for software vulnerability detection, test, and evaluation. Institute for Defense Analyses IDA Paper P-5061. Retrieved from http://www.acq.osd.mil/se/docs/P-5061-software-soar-mobility-Final-Full-Doc-20140716.pdf

[2] SAMATE. (2017). Source code security analyzers (SAMATE list of static analysis tools). Retrieved from https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

[3] MITRE. (2017, July 20). Common vulnerabilities and exposures. Retrieved from https://cve.mitre.org/

[4] Center for Assured Software, U.S. National Security Agency (2011, December). CAS static analysis tool study - Methodology. Retrieved from http://samate.nist.gov/docs/ CAS_2011_SA_Tool_Method.pdf

[5] Black, P. E., & Ribeiro, A. (2016, March). SATE V Ockham sound analysis criteria. NISTIR 8113. https://dx.doi.org/10.6028/NIST.IR.8113. Retrieved from http://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8113.pdf

[6] MITRE. (2017, June 6). Common weakness enumeration: Process: Approach. Retrieved from https://cwe.mitre.org/about/process.html#approach

[7] MITRE. (2017, June 7). Common weakness enumeration: About CWE. Retrieved from https://cwe.mitre.org/about/index.html

[8] MITRE. (2017). CVE-2009-2559. Retrieved from http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2009-2559

[9] MITRE. (2017, May 5). CWE-126: Buffer over-read. Retrieved from http://cwe.mitre.org/data/definitions/126.html

[10] MITRE. (2017, May 5). CWE- CWE-834: Excessive iteration. Retrieved from http://cwe.mitre.org/data/definitions/834.html

[11] MITRE. (2017). CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. Retrieved from http://cwe.mitre.org/data/definitions/119.html

[12] National Vulnerability Database, National Institute of Standards and Technology. (2010, August 21). CVE-2009-2559 Detail. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2009-2559

[13] Kratkiewicz, K., & Lippmann, R. (2005). Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools. Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools, 2005. Retrieved from https://www.ll.mit.edu/ mission/cybersec/publications/publication-files/full_papers/050610_Kratkiewicz.pdf

[14] SAMATE, National Institute of Standards and Technology. (2017). Software Assurance Reference Dataset. Retrieved from https://samate.nist.gov/SARD/

[15] Rutar, N., Almazan, C. B., & Foster, J. S. (2004). A comparison of bug finding tools for Java. Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04), France, November 2004. https://dx.doi.org/10.1109/ISSRE.2004.1

[16] Kupsch, J. A., & Miller, B. P. (2009). Manual vs. automated vulnerability assessment: A case study. In Proceedings of the 1st International Workshop on Managing Insider Security Threats (MIST-2009), Purdue University, West Lafayette, IN, June 15-19, 2009.

[17] Li, Z., Zou, D., Xu, S, Jin, H., Qi, H., & Hu, J. (2016). VulPecker: An automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pp. 201-213. https://dx.doi.org/10.1145/2991079.2991102

[18] De Oliveira, C., & Boland, F. (2015). Real world software assurance test suite: STONESOUP (Presentation). IEEE 27th Software Technology Conference (STC '2015) October 12-15, 2015.

[19] De Oliveira, C. D., Fong, E., & Black, P. E. (2017, February). Impact of code complexity on software analysis. NISTIR 8165. https://dx.doi.org/10.6028/NIST.IR.8165. Retrieved from http://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8165.pdf

[20] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., & Whelan, R. (2016). LAVA: Large-scale automated vulnerability addition. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, pp. 110-121. https://dx.doi.org/10.1109/SP.2016.15

[21] Pewny J., & Holz, T. (2016). EvilCoder: Automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*, pp. 214-255. https://dx.doi.org/10.1145/2991079.2991103

[22] Black, P. E. (2012). Static analyzers: Seat belts for your code. *IEEE Security & Privacy*, 10(2), 48-52. https://dx.doi.org/10.1109/MSP.2012.2

SOFTWARE ASSURANCE ADOPTION THROUGH OPEN SOURCE

By: Corbin Moyer and Patrick Hart

Software and Security engineering as a discipline is getting increased attention across the Department of Defense (DoD) as a mission enabler. Historically the DoD used an engineering approach that is independent from the type of product. Hardware and software then followed the same generic engineering principles. These principles focused on areas such as systems integration, reliability, maintainability, and test. Aspects such as lifecycle cost have risen in recent years, but awareness of secure software development as a component of that cost has yet to reach mainstream Program Management Offices (PMOs) operations. It's important to not only test functionality. Security is now a critical enabler of success. Change needs to happen.

During the early-to-mid 2000s, the industry at large was experiencing a shift in maturation of software security processes and tools. The open community saw the foundation of the non-profit Open Web Application Security Project (OWASP) Foundation in 2004 after beginning work in 2001 (1). OWASP creates and espouses freely and widely-available documents, references, tools, and general knowledge for the sake of more secure software. In the Government space, the Information Assurance Technology Analysis Center (IATAC) and the Data and Analysis Center for Software (DACS) published a Software Security Assurance State-of-the-Art Report (SOAR) in 2007 (2). This SOAR discusses definitions of secure software along with advice for development lifecycles, secure coding recommendation, metrics, and design patterns for secure software. Also in 2007, the Air Force Application Software Assurance Center of Excellence (ASACoE) was beginning to assist programs with hands-on support (3). The message was clear: secure software matters to Industry and Government.

The implication from over 10 years of think tanks and gap analysis was that change does not happen with multi-hundred-page documents of policy and governance. Change happens at the program level with actionable, reasonable, achievable steps. Many legacy software acquisitions are already on multi-year contracts where expectations are firmly established. Specialized tools are everywhere, but getting tens-of-thousands or hundreds-of-thousands of dollars for commercial tools or expert personnel has huge programmatic overhead. PMOs are frequently hard-pressed for this extra room in the budget. While these commercial solutions may be high-quality, adoption has been slow due to barriers of cost, training, and lifecycle restrictions.

Adoption of Open Solutions

The Air Force Lifecycle Management Center (AFLCMC) Cyber Systems Engineering Division (EZC) has started using and advocating for free or open source tools. EZC is the home of the Air Force Command and Control (C2) and Rapid Cyber Acquisition (RCA) Security Controls Assessor (SCA). This SCA is responsible for security assessments of all Air Force developed C2 systems and software, along with cyber-related urgent operational needs. From years of prior assessments, there was a clear pattern that PMOs had accounted for and practiced system security but had not applied that same rigor and discipline to software.

EZC has advocated the philosophy of software security as a discipline of proper engineering rigor. Engineering principles and open source tools allow for productive and collaborative conversations with existing PMO resources. Program managers resonate with using existing and readily-available resources to leverage new processes. This removes barriers and such as purchase orders and contract negotiations. It is unsurprising that PMOs start to consider adoption of open source tools as an overnight opportunity. While this a logical, pragmatic approach, it is not without special considerations. The remainder of this article will discuss EZC's recommended tools and lessons as a case study in potential benefits to existing software acquisitions.

Lessons from the Lab

A handful of open source tools have been identified and used successfully in various stages of the software development lifecycle (SDLC). These include FindBugs, OWASP Dependency Check, OWASP Zed Attack Proxy (ZAP), cppcheck, WireShark, and SonarQube. These were eventually chosen as go-to recommendations for programs in the SCA's C2 and RCA portfolios. These tools have strong organizations, transparent management, welcoming communities, and are actively updated for new threats and vulnerabilities. While the SCA's engineering team recognizes special considerations for utilizing open source tools on classified networks, these warrant a separate discussion.

It is first worth examining the licenses of these applications to make sure there are no restrictions for Government use. Most of the open source licenses examined are concerned with a single user profiting from the use of the community's tool. Using it to make one's own application better is well within scope of common licenses, with a general expectation that any improvements will propagate back to the community. A summary of the licenses for these applications can be seen in the following table. It is important to note that there are specific questions an organization may want to ask. "Can the Government contribute back?" "Can the Government make private modifications?" These are not covered here. This applies specifically to unmodified use in a development environment or lab environment.

Table 1: Tools and Licenses

Name	License	Government Restriction?	Available?
FindBugs (Univ of Maryland)	GNU GPL 3.0	No	gnu.org
Dependency Check (OWASP)	Apache 2.0	No	github (DC)
ZAP (OWASP)	Apache 2.0	No	apache.org
Cppcheck	GNU GPL 3.0	No	github.org
WireShark	GNU GPL 2.0	No	wireshark.org
SonarQube	GNU GPL 3.0		sonarqube.org

FindBugs is an application developed by the University of Maryland. Its primary purpose is finding bugs via "bug patterns" in Java applications. It breaks these down into categories like "Bad Practice", "Correctness", "Malicious Code Vulnerability", and "Dodgy Code" (4). Unlike typical static analysis tools, it analyzes Java bytecode instead of raw source code. This has the benefit of not needing the raw source code, but means that it has a higher potential for false positives. In practice, this tool is effective at noticing potentially bad patterns. However, it does require more manual curation to become truly useful. Contractors have shown minimal resistance to implementing it, but it often requires tight collaboration between

Contractor and PMO (or EZC) engineers. There are cases where this tight relationship is difficult to achieve.

As of this writing the most recent version of the tool was released in March of 2015. While the project has official sponsorship, it is not focused on regular updates. The focus is on supporting new technologies as they develop (e.g., Java 8) rather than constant patching. It has Apache Ant support for build integration and an Eclipse plugin for individual developers. This points the application once again towards the side of manual integration for a large subset of contractors.

Overall, FindBugs has had very successful use across the SCA's portfolios. Its strongest use case comes with post-development scanning rather than continuous integration. Only Ant is officially supported. Developers trying to build with tools like Maven or Gradle will require more work. However, some community support exists for these build environments. The interface of the default application is bare-bones but functional. PMO and developer engineers typically pick the basics up without a lot of explanation, provided they understand software development. For PMOs struggling to implement static analysis, FindBugs has been a considerable success. With proper expectation management, the tool performs well and has helped identify many major flaws across the portfolios.

It is a powerful tool capable of providing proxy interception, web spidering and exploration, fuzz testing, and passive scanning

OWASP Dependency Check (OWASP DC) is a tool promoted as a mature flagship product by OWASP. It serves a simple but important function. It analyzes the dependencies (such as third-party libraries) in an application to see if there are any publicly-known vulnerabilities. It currently supports Java and .NET with limited support for Python, Ruby, and Note.js (5). Dependency Check collects build information to see if there is a Common Platform Enumeration (CPE) for the dependency. If so, associated Common Vulnerability and Exposure (CVE) information is listed in an HTML-based report (5). It is still actively and frequently maintained.

The primary drawback of OWASP DC is its dependency on an Internet connection. Upon running, the machine will attempt to access the National Vulnerability Database (NVD) to download NVD data feeds (6). For applications requiring classified environments, the application cannot connect. If it's the first time the application has been run, the application cannot establish its NVD information. This means it cannot complete a scan. This creates hurdles for Government applications on a secure network such as SIPRNet. The OWASP DC documentation provides some possible workarounds such as mirroring the NVD locally. Another drawback is its interface is through the command line. It has some limited integration options, but may require manual tweaking. These problems must be solved on a case-by-case basis and may create burden for the PMO or the developer.

OWASP DC has nonetheless proven very valuable for EZC engineers, especially among older or less actively developed projects. It is surprisingly common for developers of legacy applications to not fully understand their dependencies. This is especially true of research projects that have evolved into "full" applications in the field. OWASP DC has on several occasions provided EZC with necessary information to make sure databases, frameworks, and even cryptography libraries are tracked and updated. EZC continues to use and recommend it.

OWASP ZAP is an application security scanner and penetration tester. It is a powerful tool capable of providing proxy interception, web spidering and exploration, fuzz testing, and passive scanning (7). OWASP ZAP provides results tailored for keeping web-based applications safe from attack. It is another mature flagship product for the OWASP Foundation, and it is actively maintained and developed with a strong community. It has many build integration opportunities and has a relatively clean user interface. However, new users unfamiliar with the concepts of web application vulnerabilities may find its results confusing.

In this sense the application isn't limited by features in the normal sense. It is limited in the audience that can adequately and accurately interpret and use the results. Web application vulnerability and penetration tests are traditionally led by "red teams" of specialized individuals. As web application development is less common in government systems, there are fewer software engineers that are specialized in this area of knowledge. This may mean that an attempt to run ZAP or integrate it into the software development process may still miss problems due to misconfiguration or improper understanding of the results.

With these limitations in mind, EZC has only had one developer actively insert ZAP into the development process. This implementation was specific to penetration testing, and was inserted after the build process itself. This process has proven to be a mixed blessing. Raw results couldn't be fed back to every developer. Some didn't quite understand the impact or dismissed valid results as a false positive. Careful review with EZC uncovered the need to start slowly. This means that the need for a red team may not be eliminated. Dedicated penetration testers are still valuable. However, every small step forward in cybersecurity is important. This is especially true in automated build environments. EZC continues to work with the PMO to define a good process to best utilize the results.

The fourth tool recommended by EZC is cppcheck, a static analysis tool for C and C++ code. It features many integrations including Eclipse, Jenkins, and several pre-commit hooks for version control (8). It specifically addresses errors and does not point out all bad

practices or style deviations. This has implications regarding its findings. Whereas a typical security application may over-report things that are not issues, cppcheck's "zero false positive" mentality means in practice it's more likely to not flag a real issue than to flag a non-issue. This is a good example illustrating why using two static analysis tools whenever possible is the better solution.

Cppcheck is also in large part developed by one person, with a handful of volunteers following in support. This creates an interesting point of contention for government use. While it has built a strong reputation for reliable results, there is a known single point of failure. According to public Github commits, from February 09, 2016 through Feb 11, 2017, 55.9% of all commits were from this same one individual (10). As of this writing it is still actively developed on Github. The codebase also gets automatically scanned by Coverity with all results published online publicly (9). The combination of public source code and public code analysis helps instill confidence. EZC has recommended it with no ill effect, but this is a point of contention that any potential customer should know.

Even with this knowledge available, EZC has recommended cppcheck to several applications with positive feedback. One contractor even integrated it into their build environment to continue scanning with every build, pushing results back to individual developers. The standalone interface is spartan but has a low learning curve. Result files are also easily exportable for sharing with other engineers or organizations. While cppcheck may work best in conjunction with a second tool, its results have proven reliable and useful.

With so much attention being focused on code scanning and analysis, it is easy to overlook tools like Wireshark. Wireshark is a widely-adopted packet analyzer for network capture and diagnostics. It has been a stalwart application remaining under the leadership of its original developer, Gerald Combs (11). It is currently sponsored by Riverbed, an American Networking IT company. Wireshark is in such widespread use in the security community that it has become a de facto standard for learning packet analysis.

Wireshark's strengths and weaknesses are well documented and well understood. It is primarily useful after development, during application testing. EZC has successfully used it to analyze network activity in a lab environment. This is useful for determining unwanted connections and verifying functionality. However, its user interface does heavily rely on the user understanding networking or packet analysis. As such, it has a limited use case in the software development lifecycle. While EZC is very comfortable using and recommending it, its scope of appropriate use must be determined beforehand.

The final tool for consideration is SonarQube. This is perhaps the most suited for continuous integration. SonarQube is an analyzer for finding bugs, but also provides overall "health" checks and a centralized source of security-related information for developers and managers to see and collaborate on information. It has support for several build systems like Ant and Maven, as well as continuous integration engines like Jenkins and Bamboo. SonarQube bills itself as more of a security management tool, and should be used alongside tools like cppcheck or ZAP.

SonarQube has the most modern user interface of the tools discussed so far, but it does so with relatively plain design. It performs best when used as a part of a continuous integration platform (12). However, not all DoD acquisition developers are set up to operate this way. Many developers on existing programs are still using waterfall development, or utilizing a hybrid agile-waterfall approach. They do not have continuous integration engines like Jenkins set up, nor do they have a culture of continuous fixing and patching. This, in fact, makes SonarQube difficult for adoption in this style of development. Additionally, any developer wishing to share results with a PMO may require giving a PMO Virtual Private Network (VPN) access, or save and email reports. This eliminates some of the benefits of SonarQube, and provides the PMO with no strong benefit over a program like Dependency Check.

Many developers on existing programs are still using waterfall development, or utilizing a hybrid agile-waterfall approach

With these expectations in mind, EZC frequently recommends new projects insert continuous test integration into software and security engineering from the start. SonarQube is one reliable free tool to accomplish this goal and has robust support. While it may be difficult to incorporate after, it is useful for tracking security metrics throughout development. This is an incredibly important function for modern application development regardless of what tool is used. Education is an important step in the current state of security, and EZC works closely with programs to establish security from the very start.

Recommendations

Each of these tools has a specific place in the software development lifecycle. The choice of what type of tool to use is almost as important as the choice of the tool itself. To this end, PMOs need to take a proactive approach to software security engineering based on where they are in application development. Table 2 below shows where in the development lifecycle a developer is most likely to discuss and utilize the tools outlined above. Note that a good software development lifecycle is cyclical and continuous, not linear as depicted in the table.

Table 2: Software Development Lifecycle Applications

	Planning	Analyzing	Designing	Implementing	Testing	Maintaining
FindBugs				X	X	X
OWASP DC			X	X		X
ZAP				X	X	X
Cppcheck				X	X	X
WireShark					X	X
SonarQube		X	X	X	X	X

The tools discussed in this article are heavily skewed towards the implementing, testing, and maintaining portion of the software development lifecycle. This appears to be the area in which developers traditionally receive the least amount of security training. It is also the part of the lifecycle where problems are hardest to spot by manual review. There is a trade-off between human review and automatic review, and the open source community seems to have tackled the implementation and maintenance stages first.

FindBugs, cppcheck, and SonarQube are especially useful when trying to map to the Software Engineering Institute (SEI) Secure Coding standards for the C, C++, and Java languages. These tools have many of their unique finding identifiers mapped to the recommendations and rules that comprise the SEI standards. The SEI website provides this mapping (13). EZC has found this valuable in mapping to the Risk Management Framework (RMF) and the Defense Information Systems Agency (DISA) Security Technical Implementation Guide (STIG) for Application Security and Development.

Conclusion

PMOs must use increasingly limited resources to solve security engineering problems. As such, open source tools can readily be used as a part of the development process. They are a reliable, actionable way PMOs can make their systems more secure in the short term. They can be an incredibly important piece of a robust application security program in the long run. Smart organizations recognize these tools cannot be a substitute for software security engineering. Software should still abide by principles such as following open standards, using standard interfaces, and avoiding tight coupling. Once applications have a well-thought-out design and their usage has been accounted for, these automated tools can assist developers find and fix bugs early.

Secure software development is about culture, drive, and expectation. EZC encourages the DoD at large to examine open source tools, embrace the secure software community, and share best practices. Open source tools are a great start and can be a catalyst or building block of a strong software security engineering program. Given the DoD's advanced threat landscape and large software acquisition community, we hope to see broader embracing and adoption of open source software security tools and practices.

REFERENCES

[1] About The Open Web Application Security Project. (2017, March 16). Retrieved April 04, 2017, from https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project#The_OWASP_FoundationThe SOAR itself, use the pdf to generate a citation

[2] Goertzel, K. M., et al. (2007). ITAC DACS State-of-the-Art Report. Software Security Assurance. Retrieved April 03, 2017, from http://www.dtic.mil/dtic/tr/fulltext/u2/a472363.pdf

[3] Kleffman, M., Maj. (2008). Application Software Assurance Center of Excellence (ASACOE). Retrieved April 04, 2017 from https://www.acsac.org/2008/program/case-studies/Kleffman.pdf

[4] FindBugs Bug Descriptions. (2015, March 06). Retrieved April 04 , 2017, from http://findbugs.sourceforge.net/bugDescriptions.html

[5] OWASP Dependency Check. (2017, January 23). Retrieved April 04, 2017, from https://www.owasp.org/index.php/OWASP_Dependency_Check

[6] Dependency Check. (2017, January 22). Retrieved April 04, 2017, from http://jeremylong.github.io/DependencyCheck/data/index.html

[7] OWASP Zed Attack Proxy Project. (2017, April 04). Retrieved April 04, 2017, from https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[8] Cppcheck. (2017, April 01). Retrieved April 04, 2017, from http://cppcheck.sourceforge.net/

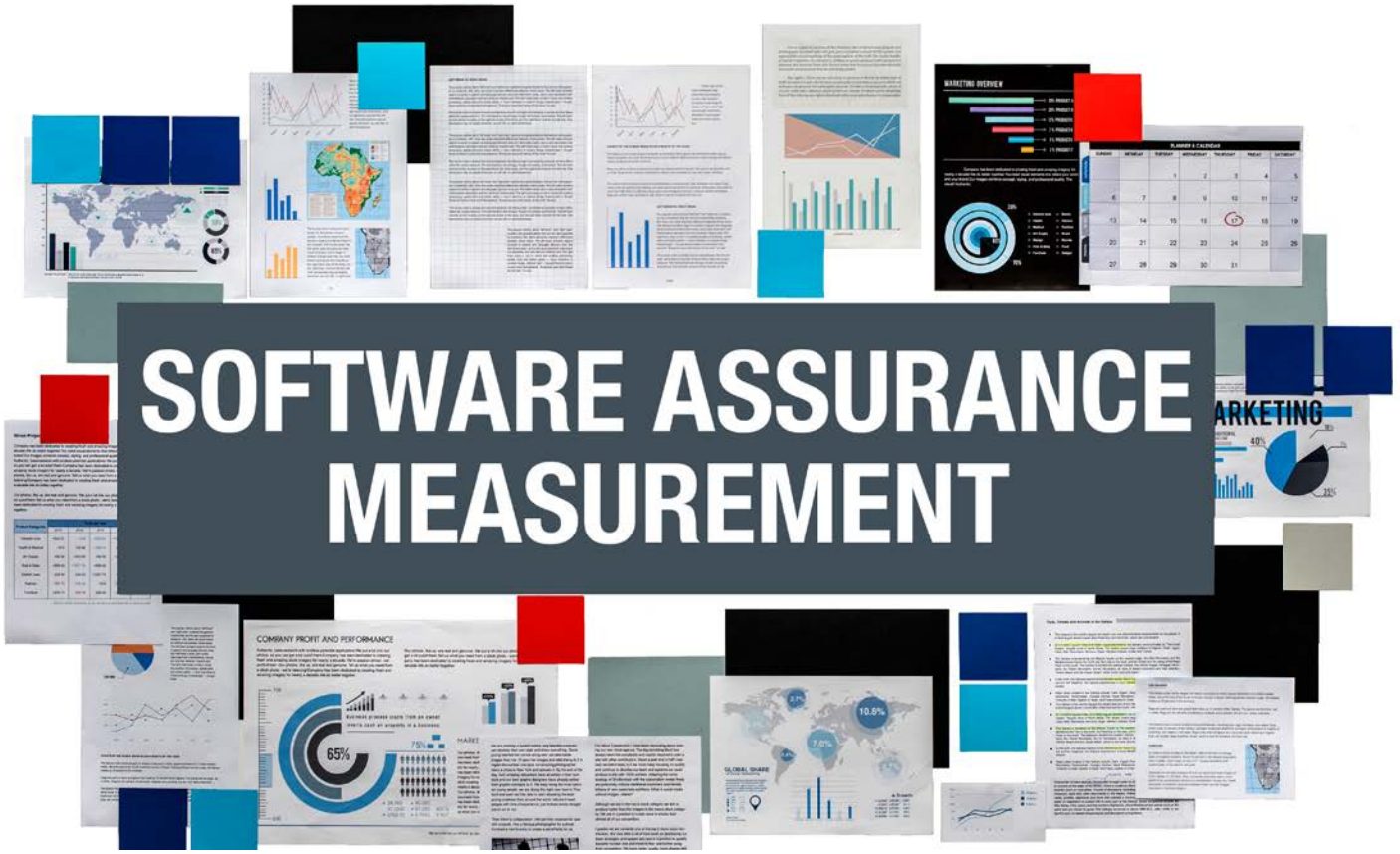
[9] Coverity Scan: cppcheck. (2017, January 17). Retrieved April 04, 2017, from https://scan.coverity.com/projects/512

[10] Contributors to danmar/cppcheck. (n.d.). Retrieved April 04, 2017, from https://github.com/danmar/cppcheck/graphs/contributors?from=2016-02-09&to=2017-02-11&type=c

[11] Wireshark. (2017, March 03). Retrieved April 04, 2017, from https://www.wireshark.org/

[12] Features. (n.d.). Retrieved April 04, 2017, from https://www.sonarqube.org/features/

[13] SEI CERT Coding Standards. (2017, March 28). Retrieved April 07, 2017, from https://www.securecoding.cert.org/confluence/display/seccode/SEI CERT Coding Standards



Establishing a Confidence that Security is Sufficient

By Dr. Carol C. Woody (SEI) and Dr. Robert J. Ellison (SEI)

Measuring the software assurance of a product as it functions within a specific system context involves assembling carefully chosen metrics that demonstrate a range of behaviors to establish confidence that the product functions as intended and is free of vulnerabilities. The first challenge is to establish that the requirements define the appropriate security behavior and the design addresses these security concerns. The second challenge is to establish that the completed product, as built, fully satisfies the specifications. Measures to provide assurance must, therefore, address requirements, design, construction, and test. We know that software is never defect free. According to Jones and Bonsignour, the average defect level in the U.S. is 0.75 defects per function point or 6,000 per million lines of code (MLOC) for a high-level language (1). Very good levels would be 600 to 1,000 defects per MLOC, and exceptional levels would be below 600 defects per MLOC. Thus, software cannot always function perfectly as intended. Additionally, we cannot establish that software is completely free from vulnerabilities based on our research, which indicates that 5% of defects should be categorized as vulnerabilities. However, we can provide sufficient measures to establish reasonable confidence that security is sufficient. Security measures are not absolutes, but we can collect information indicating that security has been appropriately addressed in requirements, design, construction, and test to establish confidence that security is sufficient.

Introduction

There is always uncertainty about a software system's behavior. Rather than performing exactly the same steps repeatedly, most software components function within a highly complex networked and interconnected system of systems that changes constantly. A measure of the design and implementation is the confidence we have that the delivered system will behave as specified. Determining that level of confidence is an objective of software assurance, which is defined by the Committee on National Security Systems (2) as

> ***Software Assurance:*** *Implementing software with a level of confidence that the software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software, throughout the lifecycle.*

At the start of development, we have a very general knowledge of the operational and security risks that might arise as well as the security behavior that is desired when the system is deployed, and we have a limited basis for establishing confidence in the behavior of the delivered system. Over the development lifecycle, as the details of the software incrementally take shape, we need to incrementally increase our confidence level to eventually confirm that the system will achieve the level of software assurance desired for the delivered system. For example, an objective of the Department of Defense (DoD) milestones reviews is to identify issues that represent risks that could adversely affect our confidence that the deployed system will function as intended.

One practice that can be performed is to analyze the software using a series of vulnerability analysis tools and remove detected vulnerabilities from the code. We know from experience that the tools will only show a small portion of the existing vulnerabilities, so how do we measure the improved confidence?

A comparison of software and hardware reliability provides some insight into challenges for managing software assurance. For hardware reliability, we can use statistical measures, such as the mean time between failures (MTBF), since hardware failures are often associated with wear and other errors that are frequently eliminated over time. The lack of hardware failures increases our confidence in a device's reliability.

As noted by the 2005 Department of Defense Guide for Achieving Reliability, Availability, and Maintainability (RAM) a lack of software defects is not necessarily a predictor for improved software reliability. The software defect exists when the software is deployed, and the failure is the result of the occurrence of an unexpected operating condition. Too little reliability engineering was given as a key reason for the reliability failures by the DoD RAM guide. This lack of reliability engineering was exhibited by

- failure to design-in reliability early in the development process
- reliance on predictions (use of relia-bility defect models) instead of con-ducting engineering design analysis

The same reasoning applies to software assurance. We need to engineer software assurance into the design of a software system. We have to go beyond just identifying defects and vulnerabilities towards the end of the lifecycle and evaluate how the engineering decisions made during design and requirements affect the injection or removal of security defects. For example, the Common Weakness Enumeration (CWE) is a list of over 900 software weaknesses that resulted in software vulnerabilities exploited by attackers. Many of them can be associated with poor acquisition or development practices.

Define the Software Assurance Target

All good engineering and acquisition starts with defined requirements, and software assurance is no different. We must define the specific software goal for the system. From the goal we can identify ways in which the engineering and acquisition will ensure, through policy, practices, verification, and validation, that the goal is addressed.

A comparison of software and hardware reliability provides some insight into challenges for managing software assurance

If the system we are delivering is a plane, then our stated software assurance goal might be "mission-critical and flight-critical applications executing on the plane or used to interact with the plane from ground stations will have low cybersecurity risk."

To establish that we are meeting this goal, a range of evidence can be collected from the following: milestone reviews, engineering design reviews, architecture evaluations, component acquisition reviews, code inspections, code analysis and testing, and certification and accreditation. Is what we have always been doing sufficient or do we need to expand current practice? We can use the Software Assurance Framework (SAF), a baseline of good software assurance practice for government engineers assembled by the SEI, to confirm completeness and identify gaps in current practices (3).

Contuined on Page 32

Build and Operat

GOAL 1: ORGANIZE

Lead and Govern

- EO 13636: Improving Critical Infrastructure Cybersecurity
- PPD 41: United States Cyber Incident Coordination
- PPD 21: Critical Infrastructure Security and Resilience
- National Strategy for Information Sharing and Safeguarding
- U.S. Int'l Strategy for Cyber
- CNSSP-24 Policy on Assured Info Sharing (AIS) for National Security Systems(NSS)
- DoDD 8000.01 Management of the DOD Information Enterprise
- DoDI 8500.01 Cybersecurity
- The DoD Cyber Strategy
- DoD Defending Networks, Systems and Data Strategy
- DoD In Enviro

GOAL 1: ORGANIZE

Design for the Fight

- SP 800-119 Guidelines for the Secure Deployment of IPv6
- Common Criteria Evaluation and Validation Scheme (CCEVS)
- CNSSP-11 Nat'l Policy Governing the Acquisition of IA and IA-Enable IT
- DFARS Subpart 208.74, Enterprise Software Agreements
- DoDD 5000.01 The Defense Acquisition System
- DoDD 7045.20 Capability Portfolio Management
- DoDD 8115.01 IT Portfolio Management
- DoDI 5000.02 Operation of the Defense Acquisition System
- DoDI 5200.44 Protection of Mission Critical Functions to Achieve TSN
- DoDI 7000.14 Financial Management Policy and Procedures (PPBE)
- DoDI 8115.02 IT Portfolio Management Implementation
- DoDI 8330.01 Interoperability of IT and National Security Systems (NSS)
- DoDI 8510.01 Risk Management Framework for DoD IT
- DoDI 8580.1 Information Assurance (IA) in the Defense Acquisition System
- RMF Knowledge Service
- MOA between DoD CIO and ODNI CIO Establishing Net-Centric Software Licensing Agreements
- DODAF (Version 2.02) DoD Architecture Framework
- CJCSI 3170.01I Joint Capabilities Integration and Development System (JCIDS)
- Joint Publication 6-0 Joint Communications System
- CNSS National Secret Fabric Architecture Recommendations
- MOA Between DoD and DHS (Jan. 19, 2017, requires CAC)

Develop the Workforce

- CNSSD-500 Information Assurance (IA) Education, Training, and Awareness
- NSTISSD-501 National Training Program for INFOSEC Professionals
- NSTISSI-4000 COMSEC Equipment Maintenance and Maintenance Training
- NSTISSI-4011 National Training Standard for INFOSEC Professionals
- CNSSI-4012 National IA Training Standard for Senior Systems Managers
- CNSSI-4013 National IA Training Standard For System Administrators (SA)
- CNSSI-4014 National IA Training Standard For Information Systems Security Officers
- NSTISSI-4015 National Training Standard for System Certifiers
- CNSSI-4016 National IA Training Standard For Risk Analysts
- DoDD 8140.01 Cyberspace Workforce Management
- DoD 8570.01-M Information Assurance Workforce Improvement Program
- DoDI 8550.01 DoD Internet Services and Internet-Based Capabilities

Partner for Strength

- SP 800-144 Guidelines on Security and Privacy in Public Cloud Computing
- SP 800-171 Protecting CUI in Nonfederal Info Systems and Organizations
- CNSSP-14 National Policy Governing the Release of IA Products/Services…
- CNSSI-1253 Security Categorization and Control Selection for Nat'l Security Systems
- CNSSI-1253F, Atchs 1-5 Security Overlays
- CNSSI-4007 Communications Security (COMSEC) Utility Program
- CNSSI-4008 Program for the Mgt and Use of Nat'l Reserve IA Security Equipment
- DoDI 5205.13 Defense Industrial Base Cyber Security / IA Activities
- DoD 5220.22-M, Ch. 2 National Industrial Security Program Operating Manual (NISPOM)
- ICD 503 IT Systems Security Risk Management and C&A

GOAL 2: ENABLE

Secure Data in Transit

- FIPS 140-2 Security Requirements for Cryptographic Modules
- SP 800-153 Guidelines for Securing Wireless Local Area Networks
- CNSSP-1 National Policy for Safeguarding and Control of COMSEC Material
- CNSSP-15 Use of Pub Standards for Secure Sharing of Info Among NSS
- CNSSP-17 Policy on Wireless Communications: Protecting Nat'l Security Info
- CNSSP-19 National Policy Governing the Use of HAIPE Products
- CNSSP-25 National Policy for PKI in National Security Systems
- NSTISSP-101 National Policy on Securing Voice Communications
- *NACSI-2005 Communications Security (COMSEC) End Item Modification*
- CNSSI-5000 Guidelines for Voice Over Internet Protocol (VoIP) Computer Telephony
- CNSSI-5001 Type-Acceptance Program for VoIP Telephones
- NACSI-6002 Nat'l COMSEC Instruction Protection of Gov't Contractor Telecomm's
- NSTISSI-7003 Protective Distribution Systems (PDS)
- DoDD 8100.02 Use of Commercial Wireless Devices, Services, and Tech in the DoD GIG
- DoDD 8521.01E Department of Defense Biometrics
- DoDI 4650.01 Policy and Procedures for Mgt and Use of the Electromagnetic Spectrum
- DoDI 8100.04 DoD Unified Capabilities (UC)
- DoDI 8420.01 Commercial WLAN Devices, Systems, and Technologies
- DoDI 8523.01 Communications Security (COMSEC)
- DoDI S-5200.16 Objectives and Min Stds for COMSEC Measures used in NC2 Comms
- *CJCSI 6510.02D Cryptographic Modernization Plan*
- *CJCSI 6510.06C Communications Security Releases to Foreign Nations*

Manage Access

- HSPD-12 Policy for a Common ID Standard for Federal Employees and Contractors
- FIPS 201-2 Personal Identity Verification (PIV) of Federal Employees and Contractors
- CNSSP-3 National Policy for Granting Access to Classified Cryptographic Information
- *CNSSP-16 National Policy for the Destruction of COMSEC Paper Material*
- CNSSI-1300 Instructions for NSS PKI X.509
- NSTISSI-3028 Operational Security Doctrine for the FORTEZZA User PCMCIA Card
- *CNSSI-4001 Controlled Cryptographic Items*
- *NSTISSI-4003 Reporting and Evaluating COMSEC Incidents*
- *CNSSI-4005 Safeguarding COMSEC Facilities and Materials, amended by CNSS-008-14*
- *NSTISSI-4006 Controlling Authorities for COMSEC Material*
- DoDI 1000.25 DoD Personnel Identity Protection (PIP) Program
- DoDI 5200.01 DoD Information Security Program and Protection of SCI
- DoDI 5200.08 Security of DoD Installations and Resources and the DoD PSRB
- DoDI 8520.02 Public Key Infrastructure (PKI) and Public Key (PK) Enabling
- DoDI 8520.03 Identity Authentication for Information Systems
- DoDM 1000.13, Vol. 1 DoD ID Cards: ID Card Life-cycle

Assure Information Sharing

- DoDI 8320.02 Sharing Data, Info, and IT Services in the DoD
- DoDI 8582.01 Security of Unclassified DoD Information on Non-DoD Info Systems
- DoD Information Sharing Strategy
- *ASD(NII)/DoD CIO Memo Use of Peer-to-Peer File Sharing Applications Across DoD*
- United States Intelligence Community Information Sharing Strategy
- CJCSI 6211.02D Defense Information System Network: (DISN) Responsibilities
- *CJCSM 3213.02C, Ch 1 Joint Staff Focal Point*

GOAL 3: ANT

Understand the

- FIPS 199 Standards for Security Categorization of Federal Info. and Info. Systems
- SP 800-60 R1 Guide for Mapping Types of Info and Info Systems to Security Categories
- SP 800-101, R1 Guidelines on Mobile Device Forensics
- DoDI S-5240.23 Counterintelligence (CI) Activities in Cyberspace

Prevent and Dela and Prevent Attacke

- FIPS 200 Minimum Security Requirements for Federal Information Systems
- SP 800-53 R4 Security & Privacy Controls for Federal Information Systems
- SP 800-61 Rev 2 Computer Security Incident Handling Guide
- SP 800-128 Guide for Security-Focused Configuration Mgt of Info Systems
- DoDI 8551.01 Ports, Protocols, and Services Management (PPSM)
- DoD O-8530.1-M CND Service Provider Certification and Accreditation Program
- CJCSM 6510.01B Cyber Incident Handling Program

ABOUT THIS

- This chart organizes cybersecuri Strategic Goal and Office of Prim Key). Double-clicking on the box authoritative source.
- *Policies in italics indicate the doc distribution or no authoritative pu currently available.*
- The linked sites are not controlle chart. We check the integrity of t you may occasionally experience problems at the source site or th document. Please let us know if longer valid.
- CNSS policies link only to the CN implemented by its website desig
- Boxes with red borders reflect re
- Note: Users of the iPad, iPhone can view this Chart but that its hy because of Apple's decision not products. For those who desire a there are apps in the iTunes stor
- For the latest version of this char ia_policychart.html. You can sig any updates to this document.

te a Trusted DoDIN

CSIAC's DoD Cybersecurity Policy Chart (Pinup)
Visit CSIAC.org to download or subscribe to receive update alerts

Cybersecurity-Related Policies and Issuances

Developed by the DoD Deputy CIO for Cybersecurity

Last Updated: August 15, 2017

Send questions/suggestions to
info@csiac.org

rspace

NIST Framework for Improving Critical Infrastructure Cybersecurity

Quadrennial Defense Review (QDR) Report

National Defense Strategy (NDS)

formation Technology nment Strategic Plan

National Military Strategy (NMS)

National Military Strategy for Cyberspace Operations (NMS-CO)

National Military Strategic Plan for the War on Terrorism

ICIPATE

Battlespace

SP 800-59
uideline for Identifying an Information System as a NSS

SP 800-92
Guide to Computer Security Log Management

NISTIR 7693
ecification for Asset Identification 1.1

ay Attackers rs from Staying

SP 800-37 R1
Guide for Applying the Risk Mgt Framework to Fed. Info. Systems

SP 800-53A R4
ssessing Security & Privacy Controls in Fed. Info. Systems & Orgs.

SP 800-124, Rev 1
Guidelines for Managing the Security of Mobile Devices in the Enterprise

CNSSAM IA 1-10, Reducing Risk of Removable Media in NSS

oDI 8530.01, Cybersecurity Activities Support to DoD Information Network Operations

oDM 5105.21V1, SCI Admin Security Manual: Info and Info Sys Security

CJCSI 6510.01F
Information Assurance (IA) and Computer Network Defense (CND)

CHART
ty policies and guidance by
nary Responsibility (see Color
directs users to the

cument is marked for limited
blic-facing hyperlink is

d by the developers of this
the links on a regular basis, but
e an error message due to
e site's decision to move the
you believe the link is no

NSS site, per restrictions
gn.
cent updates.
or iPod Touch may find they
yperlinks are inoperable,
to fully support certain Adobe
workaround for this issue,
e for less than $1.00.
t go to http://iac.dtic.mil/csiac/
n up to be alerted by e-mail to

GOAL 4: PREPARE

Develop and Maintain Trust

CNSSP-12
National IA Policy for Space Systems Used to Support NSS

CNSSP-21
National IA Policy on Enterprise Architectures for NSS

NSTISSD-600
Communications Security (COMSEC) Monitoring

CNSSI-5002, National Information Assurance (IA) Instruction for Computerized Telephone Systems

DoDD 3020.40
Mission Assurance

DoDD 3100.10
Space Policy

DoDI 8581.01
IA Policy for Space Systems Used by the DoD

DoDD 5144.02
DoD Chief Information Officer

Strengthen Cyber Readiness

SP 800-18 R1
Guide for Developing Security Plans for Federal Information Systems

SP 800-30, Rev. 1
Guide for Conducting Risk Assessments

SP 800-126 R2
SCAP Ver. 1.2

SP 800-137
Continuous Monitoring

SP 800-39
Managing Information Security Risk

DoDD 3700.01
DoD Command and Control (C2) Enabling Capabilities

DoDD S-3710.01
National Leadership Command Capability

DoDI 8560.01
COMSEC Monitoring and Information Assurance Readiness Testing

Sustain Missions

CNSSP-18
National Policy on Classified Information Spillage

CNSSP-22, IA Risk Management Policy for National Security Systems, amended by CNSS-021-13

CNSSP-300
National Policy on Control of Compromising Emanations

CNSSI-1001
National Instruction on Classified Information Spillage

CNSSI-4004.1, Destruction and Emergency Protection Procedures for COMSEC and Class. Material

CNSSI-7000
TEMPEST Countermeasures for Facilities

NSTISSI-7001
NONSTOP Countermeasures

DoDD 3020.26
Department of Defense Continuity Programs

DoDD 3020.44
Defense Crisis Management

DoDI 8410.02
NetOps for the Global Information Grid (GIG)

Defense Acquisition Guidebook
RMF for DoD IT

NSA IA Directorate (IAD) Management Directive MD-10
Cryptographic Key Protection

Color Key - OPRs

- ASD(NII)/ASD(C3I) /DOD CIO
- CNSS/NSTISS
- DISA
- DNI
- JCS
- NIAP
- NIST
- NSA
- OSD
- STRATCOM
- USD(AT&L)
- USD(C)
- USD(I)
- USD(P)
- USD(P&R)
- Other Agencies
- Recently updated box
- Expired, Update pending

AUTHORITIES

Title 10
Armed Forces
(§§2224, 3013(b), 5013(b), 8013(b))

Title 14
Cooperation With Other Agencies
(Ch. 7:§§ 141,144,145,148,149,150)

Title 32
National Guard
(§102)

Title 40
Public Buildings, Property, and Works
(Ch. 113: §§11302, 11315, 11331)

Title 44
Federal Information Security Mod. Act,
(Chapter 35)

Title 50
War and National Defense
(§§3002, 1801)

Clinger-Cohen Act, Pub. L. 104-106

UCP
Unified Command Plan
(US Constitution Art II, Title 10 & 50)

NATIONAL / FEDERAL

Computer Fraud and Abuse Act
Title 18 (§1030)

Pen Registers and Trap and Trace Devices
Title 18 (§3121 et seq.)

Stored Communications Act
Title 18 (§2701 et seq.)

Executive Order 13691
Promoting Private Sector Cybersecurity Information Sharing

Foreign Intelligence Surveillance Act
Title 50 (§1801 et seq)

Executive Order 13526
Classified National Security Information

Executive Order 13231
as Amended by EO 13286 - Critical Infrastructure Protection in the Info Age

NSD 42, National Policy for the Security of Nat'l Security Telecom and Information Systems

Executive Order 13587
Structural Reforms To Improve Classified Nets

PPD 28, Signals Intelligence Activities

NSPD 54 / HSPD 23
Computer Security and Monitoring

A-130, Management of Fed Info Resources

FAR
Federal Acquisition Regulation

Ethics Regulations

2015 National Security Strategy

National Strategy to Secure Cyberspace

NIST Special Publication 800 Series

NISTIR 7298, Rev 2, Glossary of Key Information Security Terms

CNSSD-502
National Directive On Security of National Security Systems

CNSSD-900, Governing Procedures of the Committee on National Security Systems

CNSSD-901
Nat'l Security Telecomm's and Info Sys Security (CNSS) Issuance System

CNSSI-4009
Cmte on National Security Systems Glossary

Federal Wiretap Act
Title 18 (§2510 et seq.)

OPERATIONAL

SD 527-01
DoD INFOCON System Procedures

SI 504-04
Readiness Reporting

SI 507-01
NetOps Community of Interest (NCOI) Charter

SI 701-01
NetOps Reporting

STRATCOM CONPLAN 8039-08

STRATCOM OPLANs

Computer Network Directives (CTO, FRAGO, WARNORD)

SUBORDINATE POLICY

Security Configuration Guides (SCGs)

Component-level Policy (Directives, Instructions, Publications, Memoranda)

Security Readiness Review Scripts (SRRs)

Security Technical Implementation Guides (STIGs)

Distribution Statement A: Approved for Public Release. Distribution is unlimited.

Software Assurance Framework (SAF)

The SAF defines important cybersecurity practices for the following four categories: process management, project management, engineering, and support. Each category comprises multiple areas of cybersecurity practice. In the SAF, a set of cybersecurity practices is defined for each area and relevant acquisition and engineering artifacts are documented for each of these cybersecurity practices. An evaluator can look for evidence that a cybersecurity practice has been implemented by examining the artifacts related to that practice. In the next section of this article, we will show how measurements can be linked to these cybersecurity practices. While the same approach can be applied to all four practice categories, we will focus the remainder of our efforts in this article on engineering.

Improving Assurance

As noted in the introduction, we have to go beyond just identifying defects and vulnerabilities towards the end of the lifecycle and evaluate how the engineering decisions made during design and requirements affect the injection or removal of security defects.

Justifying Sufficient Cybersecurity Using Measurement

Measurement is a mechanism for understanding and control of software processes and products and the relationships between them. It helps in making intelligent decisions that lead to improvement over time and is essential for acquisition and development management.

A formal engineering review requires more than a description of a design. Security depends on identifying and mitigating potential faults, and a design review should verify that faults associated with important operational risks have been identified and mitigated by specific design features. We need to document the rationale behind system design decisions and provide evidence that supports that rationale. Such documentation is called an assurance case. It does not imply any kind of guarantee or certification. It is simply a way to document rationale behind system design decisions. Metrics can provide evidence that justifies the assurance associated with design decisions.

Assurance case:[3] a documented body of *evidence* that provides a convincing and valid *argument* that a specified set of critical *claims* about a system's properties are adequately justified for a given application in a given environment.

If something is important, it warrants figuring out a way to measure it. Effective measurements require planning to determine what to measure and what the measures reveal. Tracking results aids understanding of whether efforts are achieving intended outcomes.

Software assurance metrics have to evaluate the engineering practices as well as the security of the product. Answers to the questions shown in Table 1 provide evidence that the engineering applied improved security.

Table 1: Engineering Questions

Effectiveness	Was applicable engineering analysis incorporated in the development practices?
Trade-offs	When multiple practices are available, have realistic trade-offs been made between the effort associated with applying a technique and the improved quality or security that is achieved (i.e., the efficiency and effectiveness of the techniques relative to the type of defect).
Execution	How well was the engineering done?
Results applied	Was engineering analysis effectively incorporated into lifecycle development?

It is essential to link a measure to a development practice. For example, product measures such as the number of defects per million lines of code (MLOC) or the output of static analysis of the source code. How should a program respond if the number of defects per MLOC appears to be too high or there is a significant number of static analysis warnings or errors reported?

We use the Goal/Question/Metric (GQM) paradigm[4] to establish a link among mission goals and engineering practices. The GQM approach was developed in the 1980's as a structuring mechanism and is a well-recognized and widely used metrics approach. For example, a top-level goal shared among all efforts to build security into software development is to identify and mitigate the ways that a software component could be compromised. Such a goal cuts across all phases of the acquisition and development lifecycles. The first challenge is to establish that the requiremen¬¬ts define the appropriate security behavior and the design addresses these security concerns. The second challenge is to establish that the completed product, as built, fully satisfies the specifications. Measures to provide assurance must, therefore, address requirements, design, construction, and test. We can identify supporting sub-goals associated with these primary acquisition/development lifecycle activities, which may be repeated at various phases across the lifecycle.

- Requirements: Manage requirements for software security risks.
- Architecture through design: Incorpo-rate security controls and mitigations to reduce security risks for design of all software components.
- Implementation: Minimize the number of vulnerabilities inserted during coding.
- Testing, validation, and verification: Test, validate, and verify software se-curity risk mitigations.

For each of these engineering sub-goals we will explore relevant practices, outputs, and metrics that could be used to establish, collect, and verify evidence. Since each project is different in scope, schedule, and target assurance, actual implemented choices will need to vary.

Sub-Goal: Requirement management sufficiently incorporates security analysis.

We should consider if we can demonstrate sufficient assurance for a requirement as we write it. For example, consider requirements that address adverse security events for an unmanned aerial vehicle (UAV). Could a non-authorized actor take control of that device or could communications with that device be disrupted? The requirement that a UAV only acts on unmodified commands received from the group station addresses the first event, and using security controls such as encryption, we can demonstrate full assurance for that requirement. We do not have engineering techniques to guarantee continued operation during a wireless network denial of service (DNS), but we can demonstrate assurance for a requirement that specifies the actions a UAV should take to protect itself during a DNS attack (4).

Practice: Security risk assessment: Conduct an engineering-based security risk analysis that includes the attack surface (those aspects of the system that are exposed to an external agent) and abuse/misuse cases (potential weaknesses associated with the attack surface that could lead to a compromise). Following Microsoft's naming convention in (5), this activity is often referred to as *threat modeling*.

Outputs: Output specificity depends on the lifecycle phase. An initial risk assessment might only note that the planned use of a commercial database manager raises a specific vulnerability risk that should be addressed during detailed design. Whereas the risk assessment associated with that detailed design should recommend specific mitigations to the development team. Testing plans should cover high-priority weaknesses and proposed mitigations.

GQM outputs represent what an acquisition wants to learn. Examples of useful outputs appear in Table 2.

Table 2: Outputs

recommended reductions in the attack surface to simplify development and reduce security risks
prioritized list of software security risks
prioritized list of design weaknesses
prioritized list of controls/mitigations
mapping of controls/mitigations to design weaknesses
prioritized list of issues to be addressed in testing, validation, and verification

We need to evaluate the output of the engineering practices. The outputs of a security risk assessment are very dependent on the experience of the participants as well as on constraints imposed by costs and schedule. Missing likely security faults or poor mitigation analysis increases operational risks and future expenses. The rework effort to correct requirement and design problems in later phases can be as high as 300 to 1,000 times the cost of in-phase correction, and undiscovered errors likely remain after that rework [10].

Practice: Conduct reviews (e.g., peer reviews, inspections, and independent reviews) of software security requirements.

Output: Issues raised in internal reviews

The analysis of the differences arising in the outputs should answer the questions shown in Table 3.

Table 3: Technical Review Analysis

What has not been done: number, difficulty, and criticality of "to be determined" (TBD) and "to be added" (TBA) items for software security requirements Where there are essential inconsistencies in the analysis and/or mitigation recommendations: number/percentage, difficulty, and criticality of the differences Where insufficient information exists for a proper security risk analysis. Examples include emerging technologies and/or functionality where there is limited history of security exploits and mitigation.

The Heartbleed vulnerability is an example of a design flaw (6). The assert function accepts two parameters: a string S and an integer N and returns a substring of S of length N. For example, assert ("that",3) returns "tha". The vulnerability occurs with calls where N is greater than then length of S such as assert("that",500) which returns a string starting with "that" followed by 496 bytes of memory data stored adjacent to the string "that". Such calls would enable an attacker to view what should be inaccessible memory locations. The input data specification that the value of N was less than or equal to the length of the string was never verified. This is CWE-135, one of the SANS Top 25 vulnerabilities[5] and should be discovered during design reviews.

Practices that support answers to the engineering questions in Table 1 (examples shown in Table 1a below) should provide sufficient evidence to justify the claim that the Heartbleed vulnerability has been eliminated.

Table 1a: Practices/Outputs for Evidence Supporting Table 1 Questions

Practice	Output
Threat modeling	Software risk analysis identifies input data risks with input verification as mitigation.
Design includes mitigation	Input data verification is a design requirement.
Software inspections show implementation	Confirms the verification of input data.
Testing	Testing plans include invalid input data. Test results show mitigation is effective for supplied inputs.

Sub-goal (Architecture through design): Incorporate security controls and mitigations to reduce security risks for design of all software components.

This sub-goal describes the security objective for the architectural and design phases of a development lifecycle. The outputs of this phase of development are shown in Table 4.

Practice: Security risk assessment as applied to the architecture and design.

Table 4: Architectural and Design Outputs

prioritized list of design weaknesses
prioritized list of controls/mitigations
mapping of controls/mitigations to design weaknesses

The assurance question is whether an acquisition should accept a developer's claim that these outputs provide sufficient security.

Potential security defects can arise with any of the decisions for how software is structured, how the software components interact, and how those components are integrated. An identification of such weaknesses has to consider both the security and functional architectures and designs. The security architecture provides security controls such as authentication, authorization, auditing, and data encryption, and the functional/system architecture describes the structure of the software that provides the desired functionality.

The functional rather than the security architecture is the more likely source of security defects. The security architecture is typically designed as an integrated unit with well-defined external interfaces. The functional architecture is increasingly likely to include commercial software with vague or unknown assurance characteristics. Commercial product upgrades can significantly change assurance requirements, interfaces, and functionality. Systems are rarely self-contained and by design accept input from independently developed and managed external systems. Such diversity increases the difficulty of identifying and prioritizing security risks and increases the importance of independent reviews.

Identifying and mitigating architectural and design weaknesses depends on a developer not only using good engineering practices and strong tools but also on that developer's understanding of the attack patterns that could be used to exploit design, architecture, and features incorporated in the proposed system.

SQL (Structured Query Language) injections provide a good instance of where attack-pattern analysis is required to mitigate exploits and how proactive application of resulting knowledge can reduce the opportunities for design vulnerabilities. Choice of a mitigation depends on the nature of required queries. Mitigation for open-ended queries likely require the use of a vetted query library. Where we have a set of well-specified queries, such as with the account-balance example, a mitigation using a parameterized query can be effective[6]. Such a mitigation would verify that the value constructed for the variable *customer_name* meets the database specification for a name.

Evidence: Many of the secure design metrics shown in Table 5 assess how well attack knowledge was incorporated in the architectural and design activities. For a security assessment to be effective, the results have to be documented and disseminated across a development team. Outputs from the developer's design activities should include reports on security risk analysis, mitigation analysis and selection, design inspections, and implementation and testing guidance. An acquisition should be able find the evidence determining assurance from that collection of development documentation.

The question for an acquirer is whether the assembled evidence demonstrates that an assurance case for a claim that SQL injections have been sufficiently mitigated could be constructed at the completion of development.

Table 5a: Practices/Outputs for Evidence Supporting Table 5 Secure Design Measures

Practice	Output
Threat modeling	Software risk analysis identifies input data risks with input verification as mitigation.
Design	Vetted library and parameterized queries are widely used and effective mitigations.
Implementation	Coding guidance is provided for chosen mitigation. Vetted library: A software scan is proposed to verify that the library has been correctly installed and used (7). Parameterized queries: Guidance to verify usage is provided for source code inspections.
Testing	Testing plans incorporate dynamic testing for SQL injections. .

Metrics: Some measures for incorporating security into the architecture and design practices are shown in Table 5.

Table 5: Secure Design Measures

Has appropriate security experience been in-corporated into design development and re-views?
Have security risks associated with security assumptions been analyzed and mitigated?
Has attack knowledge identified the threats that the software is likely to face and thereby determined which architectural and design features to avoid or to specifically incorporate?
Have mitigations for the incorporated architec-tural and design features been guided by at-tack knowledge?
Have security functionality and mitigations been incorporated into the application devel-opment guidance?
Has guidance been provided for coding and testing?

Sub-Goal: Minimize the number of vulnerabilities inserted during coding.

Practices: Differences in efficiency and effectiveness encourage the use of a combination of inspections, static analysis, and testing to remove defects from existing code (8). Testing is covered in the following section. The effectiveness of inspections is very dependent on the experience of the participants, while the effectiveness for static analysis depends on the quality of the rules that are applied and the interpretation of the output.

The choice of a practice can depend on the type of vulnerability. The design flaws should be identified during design and inspections and not by static analysis applied to the implementation. Static analysis tools can be more effective than inspections for identifying potential weaknesses in data flows that involve multiple software components for use at integration.

Outputs:

Table 6: Coding Outputs

prioritized list of coding weaknesses associated with programming languages used and application domain
static analysis associated with coding identified weaknesses
mitigation of identified coding weaknesses code inspection results

Improving security for design concentrates on proactively analyzing compromises as the design was created rather than at its completion. A developer can be more proactive with coding by creating and enforcing guidelines to eliminate a number of coding vulnerabilities. For example, buffer overflows too often occur with a subset of the text string processing functions in the C programming language. Coding guidelines can prohibit the use of that subset of functions, and C compilers can enforce those guidelines to reject code use of those functions.

Metrics: The effectiveness for static analysis is also affected by the trade-offs made by the tool designer regarding the time required for the analysis, the expert help required to support the tool's analysis, and the completeness of the analysis. Most static analysis tools use heuristics to identify likely vulnerabilities and to allow completion of their analysis within useful times. Thus static analysis is almost always incomplete and rather than showing flaws automatically, the output of such tools frequently serve as aids for an analyst to help them zero in on security-relevant portions of code so they can find flaws more efficiently.

Table 7: Static Analysis Measures

If coding guidelines are used, has static analysis that enforces those guidelines been universally applied? Are the tools used and the rules applied appropriate for the coding risks identified during the security risk analysis? If formal inspections and in-depth static analysis have been applied to only a subset of the components, does that coverage include the code that manages the critical risks identified by security risk analysis? Have the cross-component data flows with security risks been subject to static and dynamic analysis?

Sub-Goal: Test, validate, and verify software security risk mitigations.

Practice: In some instances testing is the only time that dynamic analysis applied. Some kinds of security problems are easier to detect dynamically than statically, especially problems with remote data and control flow. Testing can also supplement static analysis in confirming that the developers did not overlook some insecure programming practices.

Outputs: Output includes testing plans, outputs of tests, and analysis of the results. For example, distinguishing true vulnerabilities from those that cannot be exploited can be easier when dynamic analysis and static analysis are combined.

Security requirements can be positive in terms of what a system should do or negative in terms of what it should not do. The requirements for authentication, authorization, encryption, and logging are positive requirements and can be verified by creating the conditions in which those requirement are intended to hold true and confirming from the test results that the software meets them. A negative requirement states that something should never occur. To apply the standard testing approach to negative requirements, one would need to create every possible set of conditions, which is infeasible. Risk-based tests target the weaknesses and mitigations identified by the security risk analysis and can confirm the level of confidence we should have that the security risks have been sufficiently mitigated. Testing metrics are shown in Table 8.

Table 8: Testing Measures

What percentage of software security requirements are covered by testing? What percentage of the security risk mitigations are covered by testing? Does security testing include attack patterns that have been used to compromise systems with similar designs, functionality, and attack surfaces? Tests should have developed based threats, vulnerabilities, and assumptions uncovered by the security analysis. For example, tests could be developed to validate specific design assumptions or the interfaces with external software components. Have dynamic security analysis and security testing been applied to mitigations? Has test coverage increased in risky areas identified by the analysis? For example, a specific component, data flow, or functionality may be more exposed to untrusted inputs, or the component may be highly complex, warranting extra attention.

Assembling the Software Assurance Case

Using the Software Assurance Framework (SAF), which provides a structure of best practices, we have created a line of site from the software assurance goal to potential metrics that would provide evidence about how the goal is addressed through good software engineering practices. Each organization will need to select a starting set of evidence in which there is justification to invest time and effort, and these may vary by the type of technology product to be acquired or developed, since concerns for software assurance will vary depending on usage.

At each technical review throughout the acquisition and development lifecycle, activity progress, outputs, and metrics should be reviewed and evaluated to confirm that progress is being made to address the sufficiency for software assurance. Each review should consider the security aspects of the solution as well as the functional capabilities of the target outcome. For each engineering review, the following result should be supported by the gathered evidence:

- **Initial Technical Review (ITR).** As-sess the capability needs (including security) of the materiel solution ap-proach.
- **Alternative Systems Review (ASR).** Ensure that solutions will be cost ef-fective, affordable, operationally ef-fective, and can be developed in a timely manner at an acceptable level of software security risk.
- **System Requirements Review (SRR).** Ensure that all system re-quirements (including security) are defined and testable, and consistent with cost, schedule, risk (including software security risk), technology readiness, and other system con-straints.
- **Preliminary Design Review (PDR).** Evaluate progress and technical ade-quacy of the selected design approach.
- **Critical Design Review (CDR).** De-termine that detail designs satisfy the design requirements (including soft-ware security) established in the spec-ification and establish the interface re-lationships.

REFERENCES

[1] Jones, Caper and Bonssignour, Oliver. *The Economics of Software Quality*. s.l. : Addison-Wesley Professional, 2011.

[2] Committee on *National Security Systems. National Information Assurance (IA) Glossary CNSS Instruction* (CNSS Instruction No. 4009). Fort George G. Meade, MD : s.n., 2010.

[3] Alberts, Christopher J. and Woody, Carol C. Prototype Software Assurance Framework (SAF): Introduction and Overview. [Online] 2017. http://resources.sei.cmu.edu/library/. CMU/SEI-2017-TN-001.

[4] *Andrew Requirements and Architectures for Secure Vehicles*. Whalen, Michael W., Cofer, Darren and Gacek, Andrew. 4, 2016, IEEE Software, Vol. 33.

[5] Howard, Michael and Lipner, Steve. *The Security Development Lifecycle*. s.l. : Microsoft Press, 2006.

[6] *Heartbleed 101*. Carvalho, Marco, et al. 4, s.l. : IEEE, July-August 2014, Security & Privacy, Vol. 12, pp. 63-67.

[7] Consortium for IT Software Quality. CISQ Specifications for Automated Quality Characteristic Measures. *Consortium for IT Software Quality*. [Online] 2012. http://it-cisq.org/wp-content/uploads/2012/09/CISQ-Specification-for-Automated-Quality-Characteristic-Measures.pdf.

[8] Jones, Capers. Software Quality in 2012: A Survey of the State of the Art. *Nancook Analytics LLC*. [Online] 2012. http://sqgne.org/presentations/2012-13/Jones-Sep-2012.pdf.

[9] Kelly, Tim P. *Arguing Safety - A Systematic Approach to Safety Case Management*. Department of Computer Science Report YCST, York University. May 1999. DPhil Thesis.

[10] Davis, Noopur & Mullaney, Julia. *The Team Software Process (TSP) in Practice: A Summary of Recent Results* (CMU/SEI-2003-TR-014). Software Engineering Institute, Carnegie Mellon University, 2003. http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=6675

HERE TO SUPPORT YOUR MISSION.

Is your organization currently facing a challenging Information Technology oriented research and development problem that you need to have addressed in a timely, efficient and cost effective manner?

HOW CAN CSIAC HELP?

In a time of shrinking budgets and increasing responsibility, CSIAC is a valuable resource for accessing evaluated Scientific and Technical Information (STI) culled from efforts to solve new and historic challenges. Our CSIAC SME network includes experienced engineers and technical scientists, retired military leaders, information specialists, leading academic researchers, and industry experts who are readily available to help prepare timely and authoritative answers to complex technical inquiries.

Once submitted, the inquiry is sent directly to an analyst who then identifies the staff member, CSIAC team member, or SME that is best suited to answer the question. The completed response is then compiled and sent to the user. Responses can take up to 10 working days, though they are typically delivered sooner.

WANT TO SUBMIT A TECHNICAL INQUIRY?

The CSIAC provides up to **4 hours of Free Technical Inquiry research** to answer users' most pressing technical questions. Our subject matter experts can help find answers to even your most difficult questions.

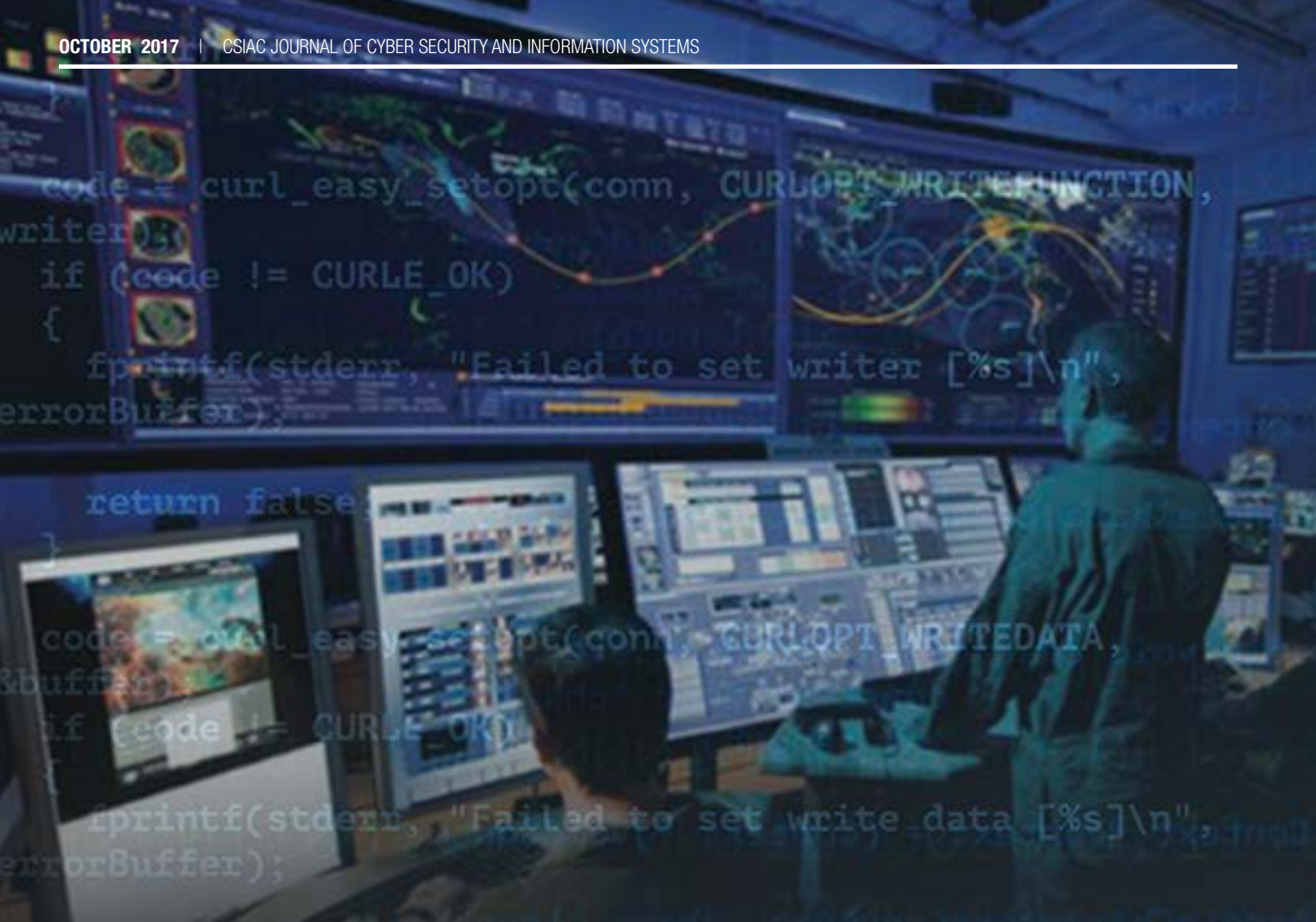
Technical inquiries can be submitted to CSIAC via our csiac.org, or by email, phone or fax.

CALL NOW! 800-214-7921

EMAIL AT: INFO@CSIAC.ORG

Cyber Security & Information Systems Information Analysis Center

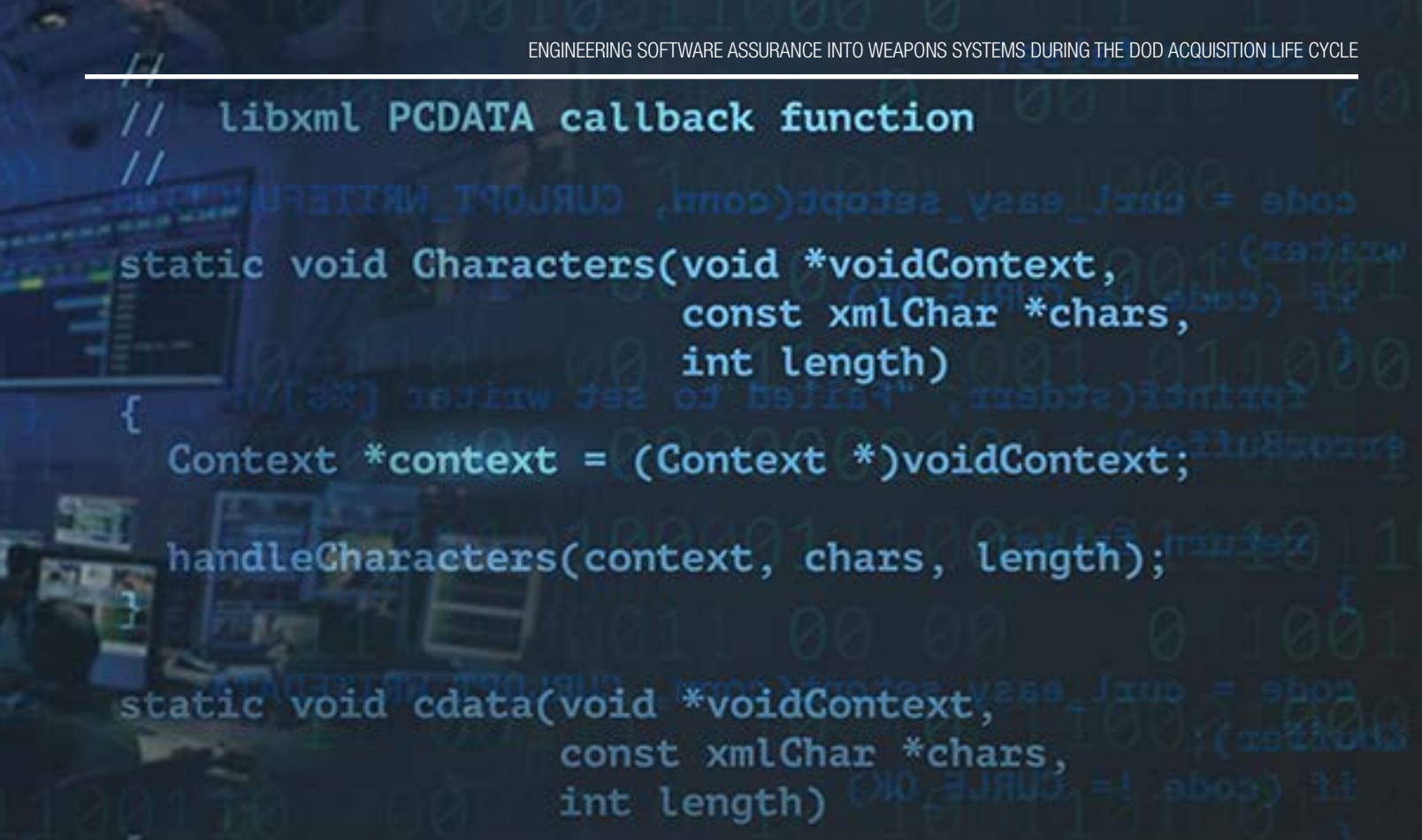
FOR MORE INFO, GO TO:
https://www.csiac.org/free-inquiries/



ENGINEERING SOFTWARE ASSURANCE INTO WEAPONS SYSTEMS DURING THE DOD ACQUISITION LIFE CYCLE

By: Dr. Scott M. Brown, Engility Corporation

Software assurance (SwA) is the "level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software, throughout the life cycle." [4] The latest change to Department of Defense (DoD) Instruction (DoDI) 5000.02, Operation of the Defense Acquisition System [1], includes a new enclosure on cybersecurity (Enclosure 14) that outlines several required actions DoD acquisition Program Managers (PMs) must implement to ensure system security and related program security across the acquisition, sustainment, and operation life cycle.



This article provides the start of a SwA user's guide, a set of recommended and tailorable "best practice" SwA activities a PM can take during development and sustainment of weapon and other systems. These best practices are based in software and systems engineering with suggested activities; expectations for conduct of Systems Engineering Technical Reviews (SETRs) with entrance and exit criteria; Program Protection Planning (PPP) considerations; and specific application of SwA tools and methods during the DoD acquisition life cycle phases. The intent is to "engineer-in" SwA into the system up front, including to system requirements using existing methods, tools, and processes and avoid attempting to "bolt on" SwA at the end of system implementation. PMs have greatly reduced latitude working with risks and vulnerabilities after system development without dramatically impacting cost, schedule and/or performance of the weapon system.

The intent is to "bake in" and engineer SwA into the system "up front and early"

Background

In February 2017 a Defense Science Board Task Force on Cyber Supply Chain summarized the need for a full life cycle approach to SwA, stating "[b]ecause system configurations typically remain unchanged for very long periods of time, compromising microelectronics can create persistent vulnerabilities. Exploitation of vulnerabilities in microelectronics and embedded software can cause mission failure in modern weapon systems.... Cyber supply chain vulnerabilities may be inserted or discovered throughout the life cycle of a system. Of particular concern are the weapons the nation depends upon today; almost all were developed, acquired, and fielded without formal protection plans." [2]

The Office of the Deputy Assistant Secretary for Defense for Systems Engineering (ODASD(SE)) leads DoD in key areas of cyber resilient systems, program protection, system security engineering (SSE), and system assurance to better understand and promote how the defense portfolio should handle evolving engineering and security challenges. The need for this focus is also reflected in National Defense Authorization Acts in recent years [3, 4, and 5] as well as in observations of programs by the Office of the Secretary of Defense (OSD), the Military Services, and defense agencies (e.g., National Security Agency, National Reconnaissance Office, and Missile Defense Agency).

Public Law 111–383, National Defense Authorization Act (NDAA) for Fiscal Year 2013, Section 932, STRATEGY ON COMPUTER SOFTWARE ASSURANCE, required the Secretary of Defense to submit a DoD strategy for assuring the security of software and software-based applications of critical systems. A key element of the strategy was to develop "[m]echanisms for protection against compromise of information systems through the supply chain or cyberattack by acquiring and improving automated tools for—(A) assuring the security of software and software applications during software development; (B) detecting vulnerabilities during testing of software; and (C) detecting intrusions during real-time monitoring of software applications." The mandated report to Congress provided the strategy, which focused on information

assurance and cybersecurity policy and guidance. The strategy included tools and techniques for test and evaluation (T&E) and for detecting and monitoring software vulnerabilities.

Public Law 112-239, NDAA for Fiscal Year 2013, Section 933, IMPROVEMENTS IN ASSURANCE OF COMPUTER SOFTWARE PROCURED BY THE DEPARTMENT OF DEFENSE, directed USD(AT&L) to develop policy that "requires use of appropriate automated vulnerability analysis tools in computer software code during the entire life cycle of a covered system, including during development, operational testing, operations and sustainment phases, and retirement."

Public Law 113-66, the NDAA for Fiscal Year 2014, Section 937, JOINT FEDERATED CENTERS FOR TRUSTED DEFENSE SYSTEMS FOR THE DEPARTMENT OF DEFENSE, directed DoD establish and charter a federation of capabilities to support trusted defense systems and ensure the security of software and hardware developed, acquired, maintained, and used by the Department. The statute stated a key charter responsibility of the federation was to set forth "the requirements for the federation to procure, manage, and distribute enterprise licenses for automated software vulnerability analysis tools." The resulting Joint Federated Assurance Center (JFAC), chartered by the Deputy Secretary of Defense, declared Initial Operational Capability (IOC) in 2016. The JFAC is a federation of DoD organizations with a shared interest in promoting software and hardware assurance in defense acquisition programs, systems, and supporting activities. The JFAC has sought to:

- Operationalize and institutionalize assurance capabilities and expert support to programs
- Organize to better leverage the DoD, interagency, and public/private sector assurance-related capabilities, and
- Influence research and development investments and activities to improve assurance technology, methodology, workforce training, and more.

lack of software assurance policy, guidance and practice within the Department results in disorganized and/or inadequate efforts

In early 2017, the Under Secretary of Defense for Acquisition, Technology, and Logistics (USD(AT&L)) updated DoDI 5000.02 to include a new Enclosure 14, "Cybersecurity in the Defense Acquisition System." The policy states in part, "Program managers, assisted by supporting organizations to the acquisition community, are responsible for the cybersecurity of their programs, systems, and information. This responsibility starts from the earliest exploratory phases of a program, with supporting technology maturation, through all phases of the acquisition. Acquisition activities include system concept trades, design, development, T&E, production, fielding, sustainment, and disposal." PMs request assistance from the JFAC such as subject matter expertise; tools, and capabilities to support program software and hardware assurance needs; knowledge; supporting software and hardware assurance contract requirements; access to state-of-the-art T&E; training; and licenses to a suite of software vulnerability analysis tools.

Technical risk management is a fundamental program management and engineering process that should be used to detect and mitigate vulnerabilities, defects, and weaknesses in SW and HW so they do not become breachable cyber vulnerabilities in deployed systems. Cyber vulnerabilities provide potential exploitation points for

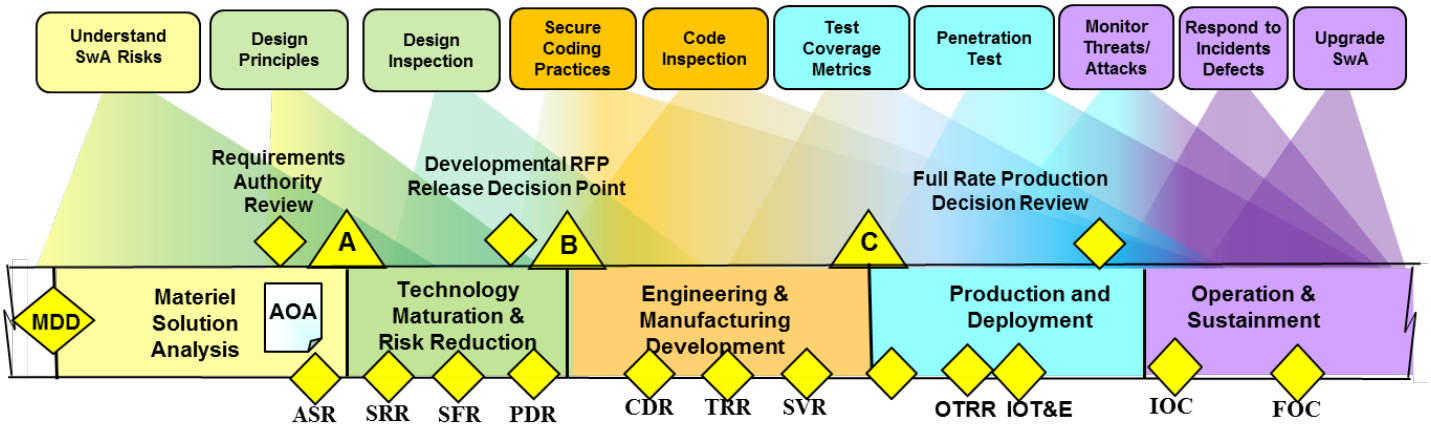


Figure 1: Software Assurance spans the entire DoD Acquisition life cycle.

adversaries to disrupt mission success by stealing, altering, or destroying system functionality, information, or technology. PMs describe in their PPPs the program's critical program information and mission-critical functions and components; the threats to and vulnerabilities of these items; and the plan to apply countermeasures to mitigate or remediate associated risks. Software is typically predominate in system functionality and SwA is one of several countermeasures that should be used in an integrated approach that also includes information safeguarding, designed-in system protections, "defense-in-depth" and layered security, supply chain risk management (SCRM) and assurance, hardware assurance, anti-counterfeit practices, anti-tamper, and program security-related activities such as information security, operations security (OPSEC), personnel security, physical security, and industrial security. SwA vulnerabilities and risk-based remediation strategies are assessed, planned for, and included in the PPP from a time frame early enough that resources can be planned and obtained.

Based on the recent DoDI 5000.02 update on cybersecurity, DoD is leading development and implementation of the supporting practices, guidance, tools and workforce competencies to ensure PMs have the ability to mitigate cybersecurity risk or vulnerabilities. Key assurance gaps exist regarding Program Management Office (PMO) activities to implement SwA within their programs as defined in the JFAC SwA Capability Gap Analysis, recently approved by the JFAC Steering Committee. This gap analysis is required by Public Law 113-66, National Defense Authorization Act (NDAA) for Fiscal Year 2014, Section 937. Where policy provides for the assessment of planning activities during development (e.g., DoDI 5000.02 mentions program support assessments and Preliminary Design Review (PDR) and Critical Design Review (CDR) assessments), software assurance should be an explicit consideration of those execution assessments.

Following are recommended SwA execution actions a PM can take during development, sustainment, and operation of weapon systems. These activities may be considered by DoD for inclusion in future policy and guidance.

Software Assurance in the DoD Acquisition Life Cycle

The Defense Acquisition Process, as provided in DoDI 5000.02, is a tailorable multi-phased development and sustainment process for all DoD programs, using six acquisition models. The phases, from Materiel Solution Analysis to Operations and Support, contain multiple milestones, decision points and technical reviews. Within this process, program management, systems engineering, T&E, and other acquisition disciplines execute their own individual but interrelated processes, and include SwA.

The development and sustainment of software is a major portion of the total system life-cycle cost, and software assurance should be considered at every phase, milestone, decision point and technical review in the acquisition life cycle both to reduce cost and to repel cyberattacks. A range of SwA activities must be planned and executed to gain assurance that any system containing software will perform operationally as expected, and only as expected. These activities blend into the entire life cycle, from requirements, to design, to implementation, to testing, to fielding, and to operation of the software. Figure 1 shows the DoD acquisition life cycle, and the tables below describe activities that should be tailored and employed among the phases and technical reviews in its process. Some of these assurance activities are also applied iteratively during the software development life cycle (not shown) whenever and wherever those software development activities occur during the DoD acquisition life cycle, such as in block, agile, or DevOps approaches.

Neglecting SwA in early life cycle activities (such as development planning, requirements, architecture assessment, design, and code development) will increase the cost of achieving assurance during later life cycle activities (such as operational testing and sustainment). But all life cycle phases require attention in the implementation of SwA. For example, thorough design and code review, use of static and origin analysis SwA tools, and follow-on remediation of findings, will both complement testing and reduce the resources expended during testing. Some flaws are more readily found through SwA tools used during review and analysis, others through dynamic analysis in testing, and certain software vulnerabilities are only detectable through manual analysis. Also the costs and benefits of specific assurance activities (e.g., code review, static code analysis, fuzz testing, and penetration testing) vary depending on the programming language, development environment, the availability of source code, the attack surface, the characteristics of the program, interoperability with other systems, and the criticality of the software in the context of the system.

Table 1 through Table 5 identify SwA considerations and specific activities associated with each phase of the acquisition life cycle. If a program is initiated later in the life cycle, for example at Milestone B, select activities from earlier phases may still be appropriate for consideration in later phases as determined by assessment of the tactical or operational use of the system compared with mission threads and system requirements. If a program is using an iterative development approach, SwA tools and methodology should be applied to individual software module development, then to integration testing and software builds so that vulnerabilities in software code are detected when they are generated, and remediated according to likelihood and consequence of adversarial attack.

The Joint Federated Assurance Center (JFAC) website (https://JFAC.army.mil), accessible by Common Access Card, provides a broad spectrum of assistance in planning and operation of assurance as an underpinning SSE activity. It also provides tools-as-a-service for all DoD programs and organizations in support of the listed activities. Four examples are assurance service providers, access to subject matter expertise, the Assessment Knowledge Base, and SwA engineering tools. The JFAC community spans DoD and can be of help at any point in the acquisition life cycle. Consider how JFAC might support a program's needs in each of the tables below.

Table 1: Software Assurance Considerations During the Materiel Solution Analysis Phase - Source: Author

SOFTWARE ASSURANCE CONSIDERATIONS (MSA Phase)
• Identify SwA roles, responsibilities, and assurance needs for the program (i.e., staffing, tools, training, etc.); plan for SwA training and resourcing. • For the risk management process, develop understanding of how the deployed system may be attacked via software and use that understanding to scope criticality and threat analyses that are summarized in the PPP. • Plan assessments and map tactical use threads, mission threads, system requirements, system interoperability, and functionality upgrades from the existing deployed system, and maintain the mapping as metadata through the last upgrade in sustainment. • Identify system requirements that may map to software and SwA requirements to facilitate trade-offs and studies to optimize functional architecture and system design, and planning and resourcing to mitigate software vulnerabilities, risks, and life cycle cost. For an integration-intensive system that relies substantially on non-development/commercial off-the-shelf (COTS)/government off-the-shelf (GOTS) items, trade space analysis can provide important information to understand the feasibility of capability and mission requirements as well as assurance of the non-developmental software supply chain. Consider alternatives to refine the system concept of implementation and optimize for modularity and digital engineering; ensure contract language for assurance reduces technical and programmatic risk. Support contracts should be part of this early solution analysis, to articulate/manage government technical data rights that later impact SwA. • Select secure design and coding standards for the program based on system functionality. • Plan and resource for the use of automated tools that determine assurance for or that detect vulnerabilities, defects, and weaknesses in requirements, allocation of requirements to functional architecture, functional architecture, allocation of functions to system design, system design, allocation of design modules to software design, coding and unit testing, and integration testing. Identify JFAC SwA service providers to assist with SwA planning and services and engage as necessary. • Develop SwA activities interconnected across the system life cycle and document in the program software engineering planning document and in the program Integrated Master Schedule (IMS).

Table 2: Software Assurance Considerations During the Technology Maturation and Risk Reduction Phase - Source: Author

SOFTWARE ASSURANCE CONSIDERATIONS (TMRR Phase)
• Incorporate SwA requirements, tool use, metrics, and assurance thresholds into solicitations. Architectures, designs, and code developed for prototyping are frequently reused later in development. • Assess system functional requirements and verification methods for inclusion of SwA tools, methodologies, and remediation across the development life cycle. • Assess requirements for SwA are correct and complete regarding assurance. Consider means of attack such as insiders and adversaries using malicious inserts; system characteristics; interoperability with other systems; mission threads; and other factors. Assure that mapping and traceability are maintained as metadata for use in all downstream assessments. • Establish baseline architecture and review for weaknesses (e.g., use of Common Weakness Enumeration (CWE)) and susceptibility to attack (e.g., use of Common Attack Pattern Enumeration and Classification (CAPEC)), and likelihood of attack success considering each detected weakness; identify potential attack entry points and mission impacts. Consider which families of automated SwA engineering tools are needed for vulnerability or weakness detection. • Review architecture and design for adherence to secure design principles and assess soundness of architectural decisions considering likely means of attack; programming language choices; development environments; frameworks; and use of open source software, etc. • Identify and mitigate technical risks through competitive prototyping while engineering in assurance. System prototypes may be physical or math models and simulations that emulate expected performance. High-risk concepts may require scaled models to reduce uncertainty too difficult to resolve purely by mathematical emulation. SW prototypes that reflect the results of key trade-off analyses should be demonstrated during the TMRR phase. These demonstrations will provide SW performance data (e.g., latency, security architecture, integration of legacy services, graceful function degradation and re-initiation, and scalability) to inform decisions as to maturity; further, EMD estimates (schedule and life cycle cost) often depend on reuse of SW components developed in TMRR; therefore to prevent technical debt, SwA considerations must have been taken into account. • Develop a comprehensive system-level architecture, then design (address function integrity, assurance of the functional breakout, function interoperation, and separation of function) that covers the full scope of the system in order to maintain capabilities across multiple releases and provide the fundamental basis to fight through cyberattack. The program focused on a given SW build/release/increment may only produce artifacts for that limited scope; however, vulnerability assessments often interact so apply system-wide and across all build/release/increment and interfaces to interoperating systems and must be maintained through development and sustainment. A PDR, for example, must maintain this system-level and longer-term, end-state perspective, as one of its functions is to provide an assessment of system maturity for the Milestone Decision Authority to assess prior to Milestone B. • Involve non-developmental item vendors in system design in order to assure functional integration addresses actual vendor product capabilities. In an integration-intensive environment, system models may be difficult to develop and fully exploit if many system components come from proprietary sources or commercial vendors with restrictions on data rights. Explore alternatives early and consider model-based systems engineering (MBSE) as the means to engineer-in assurance. Validating system performance and security assumptions may be difficult or even impossible. Proactive work with the vendor community to support model development and support informs downstream assessments including in sustainment. • Establish and manage entry and exit criteria for SwA at each SETR in order to properly focus the scope of the reviews and achieve usable assessment results and thresholds. Increasing knowledge / definition of elements of the integrated system design should include details of support and data rights.

Table 3: Software Assurance Considerations During the Engineering and Manufacturing Development Phase - Source: Author

SOFTWARE ASSURANCE CONSIDERATIONS (EMD Phase)
• Review architecture and design to assess against secure design principles; including system element / function isolation, least common mechanism, least privilege, fault isolation, graceful degradation, function re-initialization, input checking, and validation. These are the engineering basis that enable system resilience. • Enforce secure coding practices through code inspection augmented by automated static and origin analysis tools, and secure code standards for the languages used. • Detect vulnerabilities, weaknesses and defects in the software as close to the point of generation as possible, prioritize according to likelihood and consequence of use by an adversary, remediate, and regression test. • Confirm SwA requirements, vulnerability remediations, and unresolved vulnerabilities are mapped to module test cases and to the final acceptance test cases. This provides a basis for assurance that will be used in downstream assessments and system changes in sustainment. Ensure program critical function software and critical components receive rigorous automated SwA tool assessment including static code analysis (SCA), origin analysis, and penetration and fuzz testing including application of test coverage analyzers. Multiple SCA tools should be used. • Ensure CDR software documentation represents the design, performance, and test requirements, and includes development and software/systems integration facilities for coding and integrating the deliverable software and assurance operations on the integrated development environment. Software and systems used for computer software configuration item (CSCI) development (e.g., simulations and emulations) should be assured whenever possible. Problem report metadata should include assurance factors such as CWE and Common Vulnerabilities and Exposure (CVE) numbers wherever relevant so that data is usable for tracking, reporting, and assurance assessments. Legacy problem report tracking information can be used to profile and predict which types of software functions may accrue what levels of problem reports. Assessments of patterns of problem reports, or vulnerabilities among software components of the system can provide valuable information to support program resource and progress decisions. • Address systems assurance (SW, HW, FW, function, interoperability) up front and early vs. delay until later software builds. For a program using an incremental software development approach, technical debt may accrue within a given build, and across multiple builds without a plan or resources to remediate code vulnerabilities as they are generated. Technical reviews, both at the system and build levels, should have a minimum viable requirements and architecture baseline that includes SwA requirements and assured design architecture considerations, as well as ensuring fulfillment of a build-centric set of incremental review criteria and requirements that include assurance. This baseline should be retained for use through the last upgrade in sustainment. For build content that needs to evolve across builds, the PM and the systems engineer should ensure that system-level vulnerabilities, defects, and weaknesses are recorded and mitigated as soon as practical to ensure any related development or risk reduction activities occur in a timely manner. Configuration management and associated change control/review boards can facilitate the recording and management of build information and mapped assurance metadata. • Ensure all detectable vulnerabilities, defects, and weaknesses are remediated before each developmental module is checked into CM. • Install system components in a System Integration Lab (SIL) and assess continuously for assurance considerations throughout EMD. Assurance considerations include version update and CM of all COTS in the IDE, including assurance tools, operational assurance tools for the IDE, techniques using operational assurance tools to detect insider threats and malicious activity, and configuration control of the installed software and files. Details of the use of developmental system interfaces should be assessed and validated to ensure their scalability, suitability, and security for use. The emphasis in an integration-intensive system environment may be less on development and more on implementation and test. Progressive levels of integration, composition, and use should be obtained in order to evaluate ever higher levels of system performance, conducting automated penetration and fuzz testing, ultimately encompassing end-to-end testing based on user requirements and expectations. Assessment and test results should be maintained for downstream test activities and system changes. If the system is later breached, assurance metadata generated during EMD will be a basis to determine behavior, impacts, and remediations. “Glue” code and other scripted extensions to the system operational environment to enable capability should be handled in as rigorous a manner for assurance as any developed software, i.e., kept under strong configuration management, scanned with multiple SwA tools, and inspected; updates should be properly regression-tested and progressively integrated/tested

Table 4: Software Assurance Considerations During the Production and Deployment Phase - Source: Author

SOFTWARE ASSURANCE CONSIDERATIONS (P&D Phase)
• Continue to enforce secure design and coding practices for all SW changes, such as for installation modifications, through inspections and automated scans for vulnerabilities and weaknesses and maintain assessment results. • Conduct automated code vulnerability scans using SCA and origin assessment tools, reporting, and prioritization, and execute defect remediation consistent with program policy as system changes occur. Tool updates can detect additional vulnerabilities, and installations in deployment can change SW characteristics or code. • Conduct penetration testing using retained red-team or other automated test cases to detect any variations from expected system behavior. • Maintain and enhance added automated regression tests for remediated vulnerabilities, and employ test coverage analyzers to ensure sufficient test coverage for remediations. • Progressive deployment of an integration-intensive system provides infrastructure and services and higher-level capabilities as each release is verified and validated. A rigorous release process includes configuration management and the use of regression test suites that include SwA tools. The PM, systems engineer, software engineer and systems security engineer should ensure user involvement in gaining understanding and approval of changes to design, functions, or operation that may result from vulnerability remediations. • Synchronize and time block builds as much as possible to avoid forced upgrades or other problems at end-user sites. End user sites that perform their own customization or tailoring of the system installation should ensure that changes are mapped from the standard configuration, recorded, and shared with the PMO or the integrator/developer so that problem reporting and resolution activities account for any operational and performance implications, and so vulnerability assessment data are updated. Any changes should be scanned at the deployment site with multiple SwA tools to assure that no detectable vulnerabilities were inserted. This information will be necessary for assessments in detecting breaches, and for remediation of breaches.

Table 5: Software Assurance Considerations During the Operations and Support Phase - Source: Author

SOFTWARE ASSURANCE CONSIDERATIONS (O&S Phase)
• The SW sustainment activity should take ownership of all assurance-related metadata and results in support of the PMO and system operation. • Continue to enforce secure design and coding practices during sustainment through inspections and automated scans for vulnerabilities and weaknesses during sustainment system upgrades, revisions, Engineering Change Proposals, patches, and builds. System changes in sustainment can be as significant as new acquisitions. • Continue to conduct automated code vulnerability scans, reporting, and prioritization, and execute defect remediation. • Maintain and enhance automated regression tests for remediated vulnerabilities, and employ test coverage analyzers to assure sufficient test coverage. • Continue to conduct penetration testing using retained red-team or other automated test cases to detect any variations from expected system behavior • Develop and use procedures to facilitate/ensure effective software configuration management and control. For example, require static and origin analysis scans for any changes to executable scripts or code, with all detected vulnerabilities remediated before the changes are approved. A defined block change or follow-on incremental development which delivers new or evolved capability, maintenance, security, safety, or urgent builds and upgrades to the field should be accomplished using this best practice. Procedures for updating and maintaining software on fielded systems often requires individual user action, and may require specific training. There are inherent security risks involved in installing or modifying software on fielded weapon systems used in tactical activities. This should be anticipated and remediated during the MSA phase. For example, the software would have been designed so that device update in tactical situations can be assured in-situ to reduce or eliminate the opportunities for malicious insertion, corruption, or loss of software or data. Software updates to business and IT systems can also pose risks to operational availability through insider threats that should be anticipated and mitigated during the MSA phase. For example, scan glue code periodically and assess any unknown changes for malicious insertions, and scan all executable SW or scripts whenever changes are applied. For any changes that impact system function, assess the design to maintain separation of function. PMs and systems and software engineers should implement procedures and tools to assure the supply chain in order to reduce risk and prevent malicious insertions. The supply chain includes sources for COTS, GOTS, and open source libraries. • Maintain test cases previously developed for automated penetration and fuzz testing tools used during operational testing or red-team operations during system maintenance and asynchronously conduct them to detect changes in system function, operation, or timing from the baseline. Changes can be the result of undetected and operational malicious inserts by insiders. • Plan for system upgrades/updates timed to limit the proliferation of releases and therefore focus available maintenance, assurance, and support resources. In an integration-intensive environment, security upgrades, technical refreshes, and maintenance releases can proliferate, causing loss of situational awareness of assurance posture at end-user sites. Configuration management and regression testing should be used to ensure system integrity and to maintain detailed situational awareness. • Use SwA tools such as origin analysis and penetration testing to detect changes in operational configuration between the deployed site and the tested baseline.

Systems Engineering Technical Reviews (SETRs). Three reviews are particularly important to the development of all systems: the System Requirements Review (SRR), the Preliminary Design Review (PDR), and the Critical Design Review (CDR):

- The SRR ensures the system under review is ready to proceed into initial system design. It ensures all system requirements and performance requirements derived from the Initial Capabilities Document or draft Capability Development Document are defined and testable and that they are consistent with cost, schedule, risk, technology readiness, and other system constraints.
- The PDR assesses the maturity of the preliminary design supported by the results of requirements trades, prototyping, and critical technology demonstrations during the TMRR phase. The PDR establishes the allocated baseline and confirms the system under review is ready to proceed into detailed design (development of build-to drawings, software code-to documentation, and other fabrication documentation) with acceptable risk.
- The CDR assesses design maturity, design build-to or code-to documentation, and remaining risks, and establishes the initial product baseline.

Table 6 proposes success criteria for selected SETRs. These criteria have been developed through assessment of the SwA content in numerous PPPs, in feedback through JFAC from the Services and agencies, and they continue to be improved. The guidance presented in Table 6 should be tailored to the specific SETRs employed for a given acquisition program, and for the characteristics of the program.

Initiatives

Software Assurance Capability Gap Analysis: In July 2016, the DoD JFAC SwA Technical Working Group identified 63 assurance-related DoD software and systems engineering gaps that impair the effective planning and execution of SwA within the DoD acquisition and sustainment process. The gaps are organized into seven categories:

1. Life Cycle Planning and Execution;
2. SwA Technology;
3. Policy, Guidance, and Processes;
4. Resources;
5. Contracting and Legal;
6. Metrics; and
7. Federated Coordination.

The JFAC Steering Committee recently approved the congressionally mandated analysis document, published on the

Table 6: Software Assurance Success Criteria for Conduct of Technical Reviews - Source: Author

Objective	SwA Success Criteria
System Requirements Review (SRR)	
Recommendation to proceed into development with acceptable risk. Level of understanding of top-level system requirements is adequate to support further requirements analysis and design activities. Government and contractor mutually understand system requirements including (1) the preferred materiel solution (including its support concept) from the Materiel Solution Analysis (MSA) phase, (2) available technologies resulting from the prototyping efforts, and (3) maturity of interdependent systems.	• Select automated tools for design, vulnerability scan/analysis, etc. • Establish facilities, tools, equipment, staff and funding. • Confirm contractor SEMP includes SwA roles and responsibilities • Determine security requirements for programming languages, architectures, development environment, and operational environment. • Identify secure design principles to guide architecture and design decisions. • Establish processes for ensuring adherence to secure design and coding standards. • Develop plan for addressing SwA in legacy code. • Establish assurance requirements for software to deter, detect, react, and recover from faults and attacks. • Perform initial SwA reviews and inspections, and establish tracking processes for completion of assurance requirements.
Preliminary Design Review (PDR)	
Recommendation that allocated baseline fully satisfies user requirements and developer ready to begin detailed design with acceptable risk. Allocated baseline established such that design provides sufficient confidence to support 2366b certification. Preliminary design and basic system architecture support capability need and affordability target achievement.	• Determine that baseline fully satisfies user requirements • Review architecture and design against secure design principles; including system element isolation, least common mechanism, least privilege, fault isolation, input checking and validation. • Determine if initial SwA Reviews and Inspections received from assurance testing activities capture requirements appropriately. • Confirm that SwA requirements are mapped to module test cases and to the final acceptance test cases. • Establish automated regression testing procedures and tools as a core process.
Critical Design Review (CDR)	
Recommendation to start fabricating, integrating, and testing test articles with acceptable risk. Product design is stable. Initial product baseline established. Design is stable and performs as expected. Initial product baseline established by the system detailed design documentation confirms affordability/should-cost goals. Government control of Class I changes as appropriate.	• Enforce secure coding practices through Code Inspection augmented by automated Static Analysis Tools. • Detect vulnerabilities, weaknesses and defects in the software, prioritize, and remediate. • Assure chain-of-custody from development through sustainment for any known vulnerabilities and weaknesses remaining and mitigations planned. • Assure hash checking for delivered products. • Establish processes for timely remediation of known vulnerabilities (e.g. Common Vulnerability Enumeration (CVEs)) in fielded COTS components. • Ensure planned SwA testing provides variation in testing parameters, e.g. through application of Test Coverage Analyzers. • Ensure program critical function software, Critical Program Information, and Critical Components receive rigorous test coverage

JFAC website, and directed the SwA Technical Working Group to develop a strategy to address the identified gaps. This strategy is in development and the gap analysis will be published on the JFAC portal[5].

Cyber Integrator (CI): U.S. Army Aviation and Missile Research, Development, and Engineering Center (AMRDEC) conducted a one-year pilot program in an ACAT I program to include a CI into the Program Management Organization [9]. The CI is an acquisition professional with a systems engineering background charged with the holistic assessment of software assurance, anti-tamper, hardware assurance, firmware assurance and more, for planning recommendations to the Program Office, to plan and meet assurance and cybersecurity statute, policy and guidance requirements for each phase of the acquisition life cycle.

As the principal assurance and cyber advisor to the PM, the CI:

- horizontally assesses all system security disciplines to identify gaps beyond general statutory and policy compliance;
- recommends means to fully comply with statutory and policy requirements and incorporate best practices to improve overall assurance posture;
- plans PPP activities and determines costs to implement;
- informs contract language needs, performs assessments, and makes updates to improve a program's technical assurance posture;
- conducts relevant full coverage scans; and
- continuously monitors assurance activities, provides status, and maintains awareness of changing policy and guidance and derived cyber requirements.

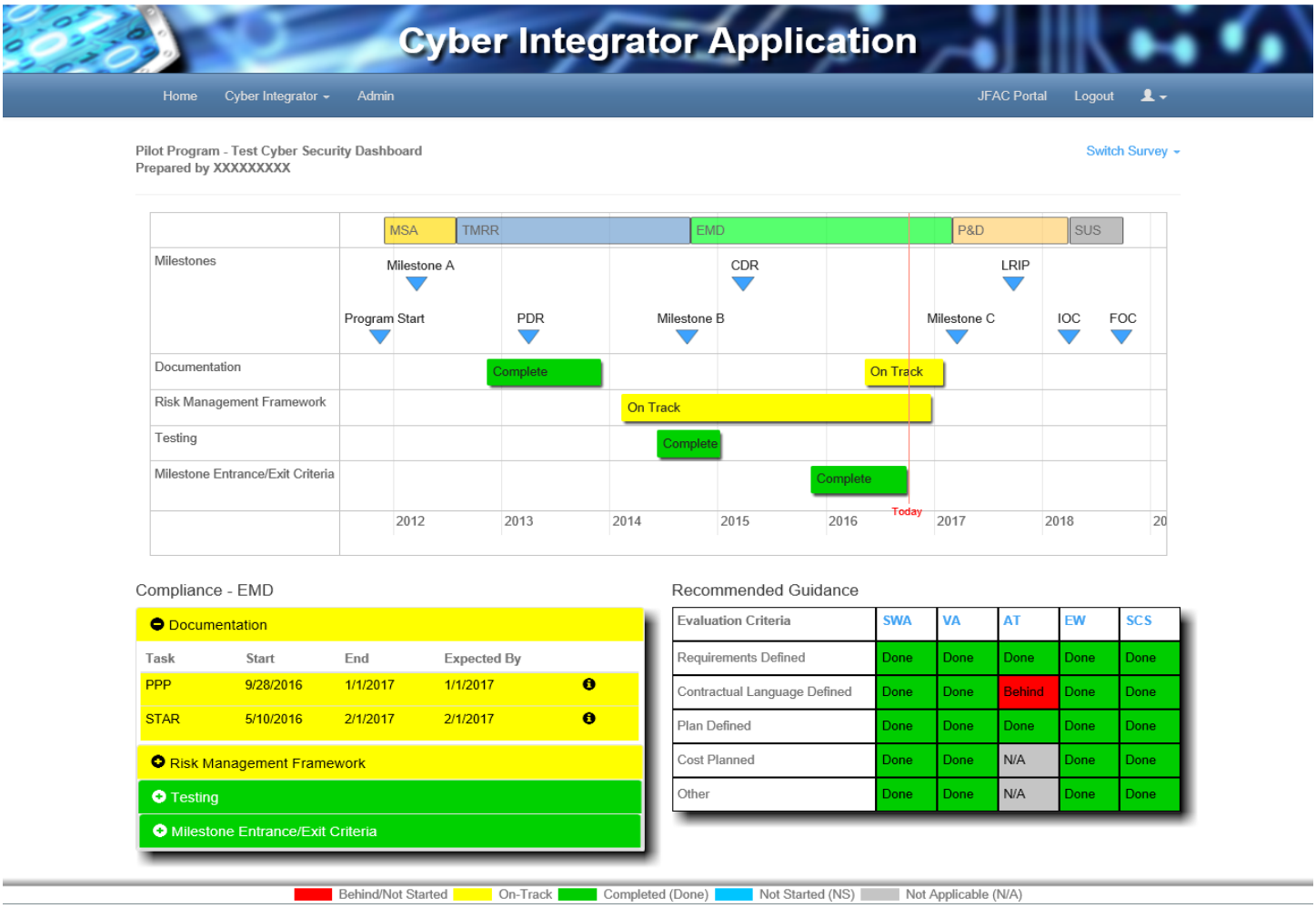


Figure 2: The Cyber Integrator Dashboard tracks cybersecurity activities across the entire DoD acquisition life cycle.

AMRDEC observed that programs normally met all compliance requirements prior to an acquisition milestone, yet with even 100% "compliance," residual assurance risk can remain in a system. AMRDEC's goal is to reduce or eliminate this residual assurance risk through implementation of system technical assurance posture, while executing best practices along the way. To aid in the conduct of the CI's responsibilities, AMRDEC, in conjunction with the JFAC, sponsored a CI Dashboard tool that supports planning, tracking and reporting assurance and cyber-related activities and requirements. The CI dashboard guides users through a series of informative survey questions to obtain appropriate assurance and compliance (statutory and policy) and recommended guidance tailored to the program. The dashboard, shown in Figure 2, is populated by the responses to the survey questions and provides the PMO an "at-a-glance" status of all the ongoing assurance and cyber activities. A more detailed report is provided for activities that are not on track, and any highlighted area can redirect the user to the specific area of insufficiency. The CI Dashboard is a pilot that will be offered as a service via the JFAC website to all DoD programs and organizations without fee or other constraint.

Conclusion

Software is the foundation of systems comprising our nation's military power. The primary and important mission capabilities of all current and foreseen weapons systems are implemented in software, and software will be 88% of the cost of DoD systems by 2024. However, the assurance that weapon system software is free of detectable vulnerabilities, defects and weaknesses that could disrupt mission, or prevent its achievement, is only now emerging as a technology and engineering discipline. The key outcome is to understand that traditional cybersecurity has a shared mission with SSE in reduction or elimination of critical vulnerabilities in the operation of weapons system and other software. Where cybersecurity has used perimeter or layered defenses to defend

systems against cyberattack, SSE builds systems from the start that are engineered to be resilient and fight through tactical cyberattack. SSE uses software assurance tools, techniques, and methodology to "engineer-in assurance" from the beginning of concept development, and throughout the system life cycle, so that vulnerabilities are discovered and fixed at the earliest possible point by the engineering team. The software becomes resilient to cyberattack, so that when the adversary penetrates cybersecurity, the mission continues. Eighty-four percent of successful cyberattacks are directly and specifically to the functions in the applications that achieve system mission, and in each case the attacks not observed by cybersecurity defenses. SSE is the discipline that will mitigate this problem. This article considered assurance of the supply chain for software, malicious insertions, latent vulnerabilities in operations, and vulnerability detection and remediation during development, sustainment, and in operations. It addressed software assurance roles and responsibilities, processes, products, tools, and software assurance capability gaps within the Department. We recommend a set of software assurance activities for each acquisition phase and technical review that a PM and their staff can tailor as appropriate to the life cycle phase and characteristics of their program. Inclusion of these activities in future authoritative guidance and tools (e.g., Cyber Integrator) will aid PMOs to execute SwA more effectively and efficiently.

REFERENCES

[1] Under Secretary of Defense for Acquisition, Technology and Logistics. *DoDI 5000.02, Operation of the Defense Acquisition System*. Instruction, Department of Defense. February 2, 2017.

[2] Office of the Under Secretary of Defense for Acquisition, Technology and Logistics. *Report of the Defense Science Board Task Force on Cyber Supply Chain*. February 2017.

[3] Public Law 111-383. National Defense Authorization Act (NDAA) for Fiscal Year 2011. Section 932, Strategy on Computer Software Assurance.

[4] Public Law 112-239. National Defense Authorization Act (NDAA) for Fiscal Year 2013. Section 933, Improvements in Assurance of Computer Software Procured by the Department of Defense.

[5] Public Law 113-66. National Defense Authorization Act (NDAA) for Fiscal Year 2014. Section 937, Joint Federated Centers for Trusted Defense Systems for the Department of Defense.

[6] Office of the Deputy Assistant Secretary of Defense for Systems Engineering. *Department of Defense Risk, Issue, Opportunities Management Guide for Defense Acquisition Programs*. Guide, Department of Defense. 2015.

[7] Department of Defense Chief Information Officer. *DoD Instruction 8510.01, Risk Management Framework (RMF) for DoD Information Technology (IT)*. Instruction, Department of Defense. 2016.

[8] Public Law 112-239. *National Defense Authorization Act for Fiscal Year 2013*. January 2, 2013.

[9] Goldsmith, Rob, and Steve Mills. *Cyber Integrator: A Concept Whose Time Has Come*. Defense AT&L Magazine. March–April 2015.

ABOUT THE AUTHORS

Mr. Thomas Hurt is the Director of the Joint Federated Assurance Center (JFAC) and lead for Software Assurance (SwA) in the Office of the Deputy Assistant Secretary of Defense for Systems Engineering (ODASD(SE)) within the Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics (OUSD(AT&L)). Before joining ODASD(SE), Mr. Hurt served as the project manager of an Army initiative to establish the use of modeling and simulation to support systems engineering trades and analyses, then automate the process to 1-day turnaround. He also led the RDECOM component of the Test and Evaluation Managers Committee for the U.S. Army Research, Development, and Engineering Command (RDECOM) Headquarters and for the Army Test Executive. Mr. Hurt founded and led TeraStore and holds more than 30 U.S. and international patents on nano-devices and their quantum electrical and magnetic effects. He worked in programs such as the TRIDENT, V-22, a global intelligence network, and the Freedom Space Station in software development and systems engineering capacities. Mr. Hurt started his career as an officer in the U.S. Marine Corps. He holds a bachelor's degree in electrical engineering from Capitol Technology University.

Scott Brown, Ph.D., serves as the technical director and deputy program manager of Engility Corporation's technical services in support of the Office of the Deputy Assistant Secretary of Defense for Systems Engineering (ODASD(SE)), Engineering Enterprise directorate. In this position, he is responsible for leading a team of over 40 personnel in the areas of Systems Engineering (SE) workforce development, specialty engineering (reliability & maintainability, manufacturing, human-systems integration, safety, value engineering), digital engineering, model-based systems engineering, modular open systems architectures, systems assurance and program protection, and policy and guidance. He is the senior subject matter expert for software assurance and the Joint Federated Assurance Center (JFAC) Coordination Center. Between 1992 and 2012, Dr. Brown served as an officer in the United States Air Force as a software engineer, acquisition professional, and squadron commander. In 2009, Lt Col (ret.) Brown was awarded the Space and Missile Systems Center Program Manager of the Year. Since 2009, Dr. Brown has directed the assessment of over 100 major defense programs, specializing in software engineering and acquisition. He earned his Ph.D. from the Air Force Institute of Technology, Wright-Patterson Air Force Base, OH, in 1998, and has published over 20 technical papers in the areas of artificial intelligence and software engineering.

THE SOFTWARE ASSURANCE *STATE-OF-THE-ART* RESOURCE

By: David A. Wheeler

Unintentional and intentionally inserted vulnerabilities in software can provide adversaries with various avenues to reduce system effectiveness, render systems useless, or even use our systems against us. Unfortunately, it can be difficult to determine what types of tools and techniques exist for evaluating software, and where their use is appropriate. The State-of-the-Art Resource for Software Vulnerability Detection, Test, and Evaluation, a.k.a. the "Software SOAR," was written to enable program managers and their staffs to make effective software assurance and software supply chain risk management (SCRM) decisions, particularly when they are developing and executing their program protection plans (PPP). A secondary purpose is to inform DoD policymakers who are developing software policies. This article summarizes the Software SOAR, including some of the over 50 types of tools and techniques available, and an overall process for selecting and using appropriate analysis tool/technique types for evaluating software. It also discusses some of the changes made in its latest update.

1. Introduction

Nearly all modern systems depend on software. It may be embedded within the system, delivering capability; used in the design and development of the system; or used to manage and control the system, possibly through other systems. Software may be acquired as a commercial-off-the-shelf (COTS) component or may be custom-developed for the system. Software is often embedded within subcomponents by manufacturers. Modern systems often perform the majority of their functions through software, which can easily include millions of lines of software code.

Although functionality is often created through software, this software can also introduce risks. Unintentional or intentionally inserted vulnerabilities (including previously known vulnerabilities) can provide adversaries with various avenues to reduce system effectiveness, render systems useless, or even turn our systems against us. Department of Defense (DoD) software, in particular, is subject to attack.

This is emphasized in the February 2017 edition of DoD Instruction 5000.02, Enclosure 14, ***Cybersecurity in the Defense Acquisition System***. This enclosure states that, "Cybersecurity is a requirement for all DoD programs and must be fully considered and implemented in all aspects of acquisition programs across the life cycle… Program managers…are responsible for the cybersecurity of their programs, systems, and information [from] the earliest exploratory phases…through all phases of the acquisition. … Program Managers will…request assistance, when appropriate, from the Joint Federated Assurance Center… to support software and hardware assurance requirements… [and] Incorporate automated software vulnerability analysis tools throughout the life cycle to evaluate software vulnerabilities…." [DoDI 5000.02 2017]

In short, analyzing DoD software to identify and remove weaknesses is a critical program protection countermeasure. What is more, because of its scale, it is often impractical to analyze software using purely manual approaches.

Unfortunately, it can be difficult to determine what types of tools and manual techniques exist for analyzing software, and where their use is appropriate. Even many software developers are unaware of the many types of tools and techniques available. Tool developers may emphasize how their tool is different from all other tools, resulting in more confusion by those trying to understand the different types of tools available.

To help reduce the confusion, our IDA team developed a document we call the "Software SOAR" or "Software Assurance SOAR" (its full title is ***State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation***). The Software SOAR was originally released to the public in 2014 [Wheeler2014], and an updated version is expected to be available soon.

The purpose of the Software SOAR is to assist DoD program managers (PM), and their staffs, in making effective software assurance (SwA) and software supply chain risk management (SCRM) decisions, particularly when they are developing their program protection plans (PPP). A secondary purpose is to inform DoD policymakers who are developing software policies.

In this article, we first highlight the overall process we recommend for selecting and reporting results from appropriate tools and techniques. This process depends on projects identifying their technical objectives, which we discuss. The next section discusses the various types of tools and techniques available to help meet those objectives, followed by a section about changes in the SOAR.

The Software SOAR includes other information not discussed here, including gaps that were identified, key topics raised in interviews, detailed fact sheets, and the impact of the mobile environment. See the Software SOAR for more information.

2. Overall Process for Selecting and Reporting Results from Appropriate Tools and Techniques

Our proposed approach for selecting various tools and techniques, and developing reports using them, is to first identify the software components in a target of evaluation (TOE) and determine each software component's context of use. Then, for each software component context of use:

1. Identify technical objectives based on context.
2. Select tool/technique types needed to address the technical objectives, using the matrix discussed below.
3. Select specific tools and techniques of the relevant types.
4. Summarize selection (write down your plan), which may be part of a larger report. Within DoD, this would be part of the PPP.
5. Apply the analysis tools, use their results, and report appropriately. Here, the selected tools and techniques are applied, including the selection, modification, or risk mitigation of software based on tool/technique results, and reports are provided to those with oversight authority.

Since different tool/technique types are better at addressing different technical objectives, we suggest ensuring that the set of tools/techniques selected adequately cover the intended technical objectives. One way to do this is to use a matrix we have developed that specifies the technical objectives met, to some degree, by various tool/technique types. The table below illustrates this matrix. The table cells indicate the applicability, e.g., a checkmark with a yellow cell indicates that the tool/technique type can be a highly cost-effective measure to address this technical objective and should be investigated further. A green circled checkmark indicates the tool/technique type completely addresses this technical objective (unfortunately, this is rare).

Technical Objective	Lower-level technical objective	Need indicator	Tool/technique type							
			Static			Dynamic			Hybrid	
			1	2	…	21	22	…	31	…
Design & code quality										
Counter unintentional-like known vulnerabilities	…			√						
Authentication & access control	Authentication									
	…						√			
Counter unintentional-like weaknesses	Buffer handling	C/C++/ Objective-C								
…	…									

3. Technical Objectives

Different types of tools and techniques are better for different purposes. Thus, it is important to identify the various purposes for using different types of tools and techniques, so that the most appropriate types can be selected. The Software SOAR terms these purposes "technical objectives."

It is common for security issues to be categorized as being related to confidentiality, integrity, and availability; DoD also separately considers authentication and non-repudiation [DoDI 8500.01]. However, since a vulnerability can cause problems in all of those areas, these categorizations are too general to support narrowing the selection of appropriate tool/technique types.

Even at a more detailed level, there is no universally accepted set of categories for technical objectives. The Common Weakness Enumeration (CWE) identifies a very large set of common weaknesses in software that may lead to vulnerabilities, but while CWE is useful for many purposes, it does not provide a single, simple organizational structure. "Top" lists, such as the "CWE/SANS top 25" and the "Open Web Application Security Project (OWASP) top 10," are helpful in identifying especially common weaknesses, but they make no attempt to cover all relevant objectives.

Instead, we have focused on identifying a set of detailed technical objectives that can help narrow the selection of appropriate tools and techniques. We created this set of technical objectives by merging several accepted sources. Here is the top-level set of technical objectives:

1. Provide design and code* quality
2. Counter unintentional-like known vulnerabilities
3. Ensure authentication and access control*
   a. Authentication Issues
   b. Credentials Management
   c. Permissions, Privileges, and Access Control
   d. Least Privilege
4. Counter unintentional-"like" weaknesses
5. Counter intentional-"like"/malicious logic*
   a. Known malware
   b. Not known malware
6. Provide anti-tamper and ensure transparency
7. Counter development tool inserted weaknesses
8. Provide secure delivery
9. Provide secure configuration
10. Other

1. Buffer Handling*
2. Injection* (SQL, command, etc.)
3. Encryption and Randomness*
4. File Handling*
5. Information Leaks*
6. Number Handling*
7. Control flow management*
8. Initialization and Shutdown [of resources/ components]*
9. Design Error
10. System Element Isolation
11. Error Handling* a Fault isolation
12. Pointer and reference handling*

4. Types of Tools and Techniques

There is no widely accepted complete categorization of tools and techniques. For example, the National Institute of Standards and Technology Software Assurance Metrics and Tool Evaluation (NIST SAMATE) project web page has a brief but limited list of tool categories. This lack of categorization is one reason why the space is so confusing.

We have created a categorization of tools and techniques based on our own analysis, using sources such as interviews and the NIST SAMATE project. It is not the only possible categorization, and since it is incomplete, we do not call it a taxonomy. Our goal is simply to create a useful set of categories that can be extended as required. In general, we only included tool types where there is at least one commercially available tool; we granted some exceptions in the mobile space because that is a fast-paced environment. We expect that new types of tools and technologies could be added in the future to these categories, driven by innovation and commercialization (especially in the mobile environment).

The latest version of the SOAR identifies 59 types of tools and techniques available for analyzing software. We have identified the following three major groups of tool/technique types:

- Static analysis: Examines the system/software without executing it, including examining source code, bytecode, and/or binaries.
- Dynamic analysis: Examines the system/ software by executing it, giving it specific inputs, and examining results and/or outputs.
- Hybrid analysis: Tightly integrates static and dynamic analysis approaches; for example, test coverage analyzers use dynamic analysis to run tests and then use static analysis to determine which parts of the software were not tested. This grouping is used only if static and dynamic analyses are tightly integrated; a tool or technology type that is primarily static or primarily dynamic is put in those groupings instead.

The following sections identify a subset of tool/technology types in each of these three major groups.

4.1. Static Analysis

Here are some of the common static analysis tool/technology types:

1. Attack modeling. Attack modeling analyzes the system architecture from an attacker's point of view to find weaknesses or vulnerabilities that should be countered.
2. Source code analyzers[1] is a group of the following tool types:
   a. Warning flags. Warning flags are mechanisms built into programming language implementations and platforms that warn of dangerous circumstances while processing source code.
   b. Source code quality analyzer. Source code quality analyzers examine software source code and search for the implementation of poor coding or certain poor architecture practices, using pattern matches against good coding practices or mistakes that can lead to poor functionality, poor performance, costly maintenance, or security weaknesses, depending on context. There is now a preponderance of evidence that higher-quality software (in general) tends to produce more secure software [Woody 2014]. These kinds of tools are often less expensive than some other kinds, and can often be applied earlier in development, providing good reasons to use them even when the focus is to develop secure software.
   c. Source code weakness analyzer. Source code weakness analyzers examine software source code and search for vulnerabilities, using pattern matches against well-known common types of vulnerabilities (weaknesses). This kind of tool is also called a "source code security analyzer," "static application security testing" (SAST) tool, "static analysis code scanner," or "code weakness analysis tool." We've chosen the name "source code weakness analyzer" because this name more clearly defines what this type of tool does and distinguishes it from other types of analysis.
   d. Context-configured source code weakness analyzer. This configures a source code weakness analyzer specifically for the product being evaluated (e.g., by adding many additional rules).
3. Binary/bytecode analysis is a group of the following tool types:
   a. Traditional virus/spyware scanner. Traditional virus/spyware scanners search for known malicious patterns in the binary or bytecode.
   b. Quality analyzer. Binary/bytecode quality analyzers examine the binary or bytecode (respectively) and search for the implementation of poor coding or certain poor architecture practices, using pattern matches against good coding practices or mistakes that can lead to poor functionality, performance, costly maintenance, or security weaknesses depending on context.
   c. Bytecode weakness analyzer. Bytecode weakness analyzers examine binaries and search for vulnerabilities, using pattern matches against well-known common types of vulnerabilities (weaknesses). Note that these are similar to source code weakness analyzers, except that the analysis is performed on bytecode.
   d. Binary weakness analyzer. Binary weakness analyzers examine binaries and search for vulnerabilities, using pattern matches against well-known common types of vulnerabilities (weaknesses). Note that these are similar to source code weakness analyzers, except that the analysis is performed on a binary.
4. Human review. This is typically done with source code, but it can also be done with binary or bytecode (often this is generated by a binary or bytecode disassembler, as noted above). Note that human reviews can apply to products other than code, including requirements, architecture, design, and test artifacts. Human reviews include the following more-specific types of techniques:
   a. Focused manual spot check. This specialized technique focuses on manual analysis of code (typically less than 100 lines of code) to answer specific questions. For example, does the software require authorization when it should? Do the software interfaces contain input checking and validation?
   b. Manual code review (other than inspections). This specialized technique is the manual examination of code, e.g., to look for malicious code.
   c. Inspections (Institute of Electrical and Electronics Engineers (IEEE) standard). IEEE 1028 inspection is a systematic peer examination to detect and identify software product anomalies.
   d. Generated code inspection. This technique examines generated binary or bytecode to determine that it accurately represents the source code. For example, if a compiler or later process inserts malicious code, this technique might detect it. This is usually a spot check and not performed across all of the code.
5. Secure platform selection is a group of the following tool types:
   a. Safer languages. This is selecting languages, or language subsets, that eliminate or make it more difficult to inadvertently insert vulnerabilities. This includes selecting memory-safe and type-safe languages.
   b. Secure library selection. Secure libraries provide mechanisms designed to simplify developing secure applications. They may be standalone or be built into larger libraries and platforms.
   c. Secured operating system (OS). A secured OS is an underlying operating system and platform that is hardened to reduce the number, exploitability, and impact of vulnerabilities.
6. Origin analyzer. Origin analyzers are tools that analyze source code, bytecode, or binary code to determine their origins (e.g., pedigree and version). From this information, some estimate of riskiness may be determined, including the potential identification of obsolete/vulnerable libraries and reused code.
7. Digital signature verification. Digital signature verification ensures that software is verified as being from the authorized source (and has not been tampered with since its development). This typically involves checking cryptographic signatures.
8. Configuration checker. Configuration checkers assess the configuration of software to ensure that it meets requirements, including security requirements. A configuration is the set of settings that determine how the software is accessed, is protected, and operates.

1 For the purposes of this paper, "source code analyzer" is a group of tool types; the lettered items below are the tool/technique types. A person who performs manual review of source code could also be considered a "source code analyzer," but for our purposes we group manual review processes separately.

4.2. Dynamic Analysis

Here are some of the common dynamic analysis tool/technology types:

1. Application-type-specific vulnerability scanner. An application-type-specific vulnerability scanner sends data to an application, to identify both known and new vulnerabilities. It may look for known vulnerability patterns (a.k.a. weaknesses) and anomalies. This is a group of the following tool types:
   a. Web application vulnerability scanner. A web application vulnerability scanner automatically scans web applications for potential vulnerabilities. They typically simulate a web browser user, by trawling through URLs and trying to attack the web application. For example, they may perform checks for field manipulation and cookie poisoning [SAMATE].
   b. Web services scanner. A web services scanner automatically scans a web service (as opposed to a web application), e.g., for potential vulnerabilities. [SAMATE]
   c. Database scanner. Database scanners are specialized tools used specifically to identify vulnerabilities in database applications. [SAMATE] For example, they may detect unauthorized altered data (including modification of tables) and excessive privileges.
2. Fuzz tester. A fuzz tester provides invalid, unexpected, or random data to software, to determine whether problems occur (e.g., crashes or failed built-in assertions). Note that many scanners (listed above) use fuzz testing approaches.
3. Automated detonation chamber (limited time) automatically isolates a program (including running multiple copies in virtual machines), executes it, detects potentially malicious or unintentionally vulnerable activities, and then reports its findings prior to the software's deployment. In contrast, we use the broader term "monitored execution" to refer to broader processes that use many tools/techniques (including manual techniques) to isolate software and detect malicious activities.

4.3. Hybrid Analysis

Here are some of the common hybrid analysis tool/technology types:

1. Test coverage analyzer. Test coverage analyzers are tools that measure the degree to which a program has been tested (e.g., by a regression test suite). Common measures of test coverage include statement coverage (the percentage of program statements executed by at least one test) and branch coverage (the percentage of program branch alternatives executed by at least one test). Areas that have not been tested can then be examined, e.g., to determine whether more tests should be created or whether that code is unwanted.
2. Hardening tools/scripts. This type of tool modifies software configuration to counter or mitigate attacks, or to comply with policy. In the process, it may detect weaknesses or vulnerabilities in the software being configured.
3. Execute and compare with application manifest. Run an application with a variety of inputs to determine the permissions it tries to use, and compare that with the application permission manifest.
4. Track sensitive data. Statically identify data that should not be transmitted or shared (e.g., due to privacy concerns or confidentiality requirements), then dynamically execute the application, tracking that data as tainted to detect exfiltration attempts.
5. Coverage-guided fuzz tester. Use code coverage information to determine new inputs to test.

5. Changes to the SOAR

In the May 2014 version of the SOAR, we noted that there was a lack of specific quantitative data to support the hypothesis that higher software quality tends to produce more secure software. At the time this was a plausible hypothesis that a number of experts believed to be true. However, many seemingly reasonable hypotheses are false. We believed in 2014 that it was important to investigate this claim before recommending it. This question is important, because if it is true, then it might be appropriate to first use tools to identify quality problems, fix the problems they identify, and then use other tools for more complex analysis. More evidence that supports this hypothesis has since been published. In particular, SEI [Woody 2014] published in December 2014 a compendium of evidence to support the claim that higher quality software (in a general sense) tends to produce more secure software.

While more evidence would be welcome, we believe the preponderance of evidence now is that improving the general quality of software tends to improve the security of the software. This does not mean that using only generic quality tools is enough to develop secure software. Instead, it means that using generic quality tools can be a valuable aid in developing secure software.

We searched for new types of tools and techniques in the commercial software market (both open source software and proprietary software). Software assurance is not a solved problem, and while most tool suppliers had refined their tools further, we were disappointed that we did not find more new approaches. That said, we added some new tool categories not in previous versions of the SOAR. For example, we added "coverage-guided fuzz tester" as a category to cover tools such as American Fuzzy Lop (an open source software tool that has found a large number of vulnerabilities).

All of these additional types of tools are hybrid approaches, which is interesting because we had previously predicted that more tool types would be created as hybrids to take advantage of the information available from both static and dynamic analysis.

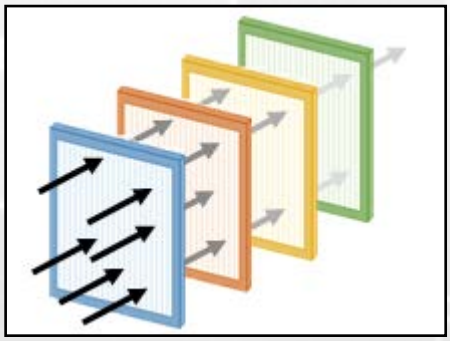
We also added more guidance on how to apply this, including tips on selecting technical objectives and how to select tools and techniques given those technical objectives. We added a mapping to the OWASP top 10 of 2013 (which is widely used when developing

web applications). We also discussed other kinds of tools that are related but not the primary focus, such as SwA correlation tools.

6. Combining Approaches

No one type of tool or technique can address all possible technical objectives. Some tool/technique types address only one or a few specific technical objectives, but are highly effective for that scope. Those that have broader applicability may have challenges (e.g., some can be more costly or require deeper expertise).

Thankfully, static, dynamic, and hybrid analysis tools and techniques can be combined to alleviate some of these limitations. The following figure is a conceptual illustration of the advantages of using multiple tools and techniques, particularly when they use different approaches. The arrows represent potential risks, including exposed vulnerabilities in the software, and the screens represent tools and techniques applied by a project. No one tool or technique addresses all technical objectives, and almost all find only a fraction of the vulnerabilities and other issues they address. Each tool or technique contributes to meeting technical objectives (and thus reducing overall risk).



However, achieving the desired result will not happen by accident. Programs need to proactively determine their technical objectives, determine what kinds of tools and techniques will help them achieve their objectives, and then smartly apply these tools and techniques. We hope that the Software SOAR will help programs identify and achieve their objectives.

REFERENCES

[1] [DoDI 5000.02 2017] DoD. February 2, 2017 (Change 2). Operation of the Defense Acquisition System. DoD Instruction 5000.02. http://www.dtic.mil/whs/directives/corres/pdf/500002_dodi_2015.pdf

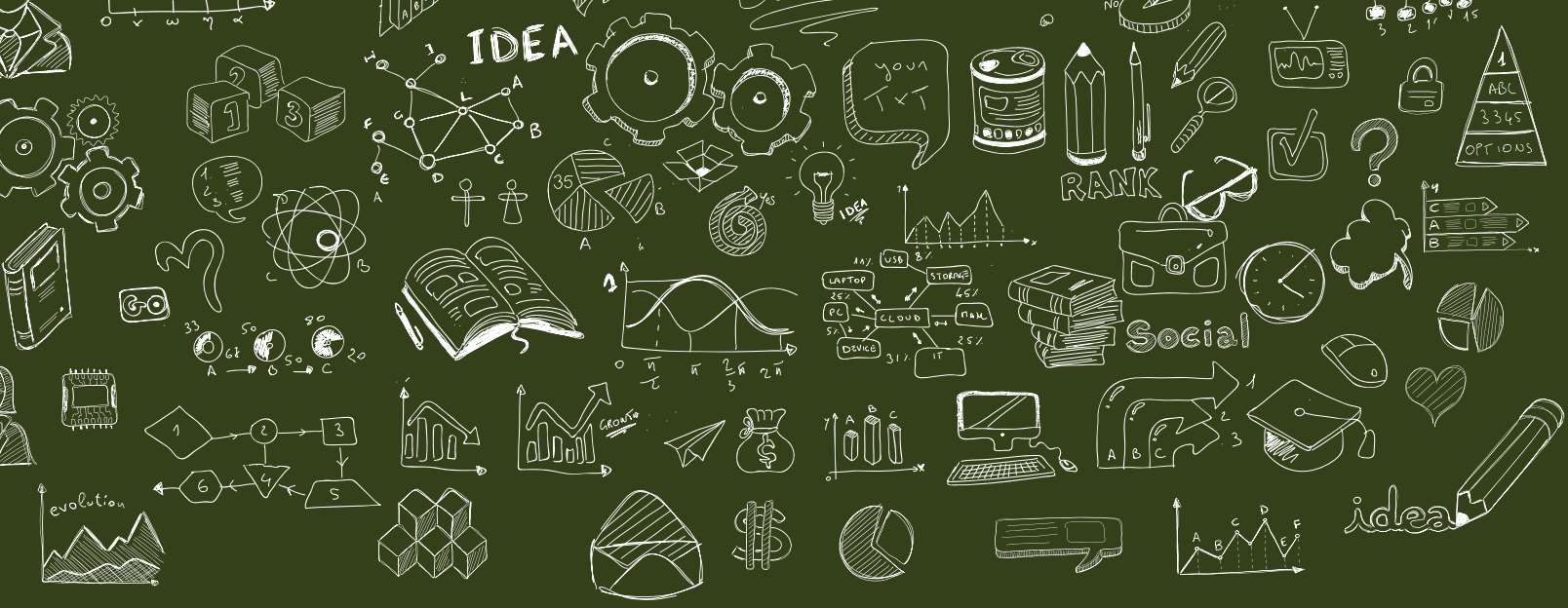
[2] [DoDI 8500.01] DoD. March 14, 2014. Cybersecurity. DoD Instruction 8500.01. http://www.dtic.mil/whs/directives/corres/pdf/850001_2014.pdf

[3] [SAMATE] "Classes of Tools & Techniques." Retrieved 2017-04-04. https://samate.nist.gov/index.php/Tool_Survey.html

[4] [Wheeler2014] Wheeler, David A. and Rama S. Moorthy. State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation. July 2014. IDA Paper P-5061. http://www.acq.osd.mil/se/docs/P-5061-software-soar-mobility-Final-Full-Doc-20140716.pdf

[5] [Woody2014] Woody, Carol, Robert Eillison, and William Nichols. "Predicting Software Assurance Using Quality and Reliability Measures." December 2014. Technical Note CMU/SEI-2014-TN-026. Carnegie Mellon University (GMU) Software Engineering Institute (SEI) CERT Division/SSD.

THIS WORK WAS CONDUCTED UNDER CONTRACT HQ0034-14-D-0001, TASK AU-5-3856, "ENHANCING PROGRAM PROTECTION THROUGH EFFECTIVE SYSTEMS ASSURANCE," FOR OFFICE OF THE DEPUTY ASSISTANT SECRETARY OF DEFENSE FOR SYSTEMS ENGINEERING; ACQUISITION TECHNOLOGY AND LOGISTICS. THE PUBLICATION OF THIS IDA MEMORANDUM DOES NOT INDICATE ENDORSEMENT BY THE DEPARTMENT OF DEFENSE, NOR SHOULD THE CONTENTS BE CONSTRUED AS REFLECTING THE OFFICIAL POSITION OF THAT AGENCY. THE MATERIAL MAY BE REPRODUCED BY OR FOR THE U.S. GOVERNMENT PURSUANT TO THE COPYRIGHT LICENSE UNDER THE CLAUSE AT DFARS 252.227-7013 (A)(16) [JUN 2013].



PILOTING SOFTWARE ASSURANCE TOOLS IN THE DEPARTMENT OF DEFENSE

By: Dr. Thomas P. Scanlon and Timothy A. Chick, Software Engineering Institute, Carnegie Mellon University

In this article, we present and describe the JFAC Enterprise Software Licensing Pilot program activities during the 2016 fiscal year. During this period, JFAC provided limited quantities of Software Assurance tools to users in the DoD with an aim of evaluating how the use of these tools could improve the state of software assurance within the Department. The four Software Assurance tools procured and made available to the end-users consisted of a dynamic web testing tool, two static source code analyzers and an origin analysis tool that checked for vulnerabilities in third party libraries and components used to build software. After the licenses had been available to the licensees for nearly a full calendar year, researchers from the Software Engineering Institute (SEI) at Carnegie Mellon University conducted an outreach and survey effort to solicit feedback on user experiences. The results of the tools' effectiveness are presented as well as findings on the impact use of the tools had on the software development process.

JFAC Enterprise Software Licensing Program Objectives

To increase the security posture of Department of Defense (DoD) software systems, the employment of Software Assurance techniques needs to shift from being part of a step in the development process to being an integrated element of the entire development process. Specifically, Software Assurance must be engineered into the entire lifecycle of applications and not just be something that is checked in a phase before deployment. The inclusion of Software Assurance testing earlier in the development process in parallel with development efforts, often referred to as "Shift Left", is keenly effective when frequent and automated tests are performed during development phases to provide timely feedback to developers.

To promote a more mature approach to Software Assurance within the DoD, the Joint Federated Assurance Center (JFAC) established an Enterprise Software Licensing Pilot program in 2016. The goals of this program are to provide enterprise-wide licenses to the DoD development community for Software Assurance tools, to promote wider use of such tools, to provide training and expertise to engineers and developers on how and when to best use these tools, and to simplify the acquisition of Software Assurance tools by systems and software engineers [JFAC 2016].

During the first year of the JFAC Enterprise Software Licensing Pilot program, JFAC provided limited quantities of Software Assurance tools to users in the DoD with an aim of evaluating how the use of these tools could improve the state of software assurance within the Department. Specifically, during this pilot phase, JFAC procured limited quantities of four commercially available Software Assurance tools and provided them to selected DoD constituents at no direct cost to the product users. The product users of these tools were largely members of the Army, Navy, and Air Force in a near even distribution and a small number of selected other DoD personnel.

The four Software Assurance tools procured and made available to the end-users consisted of a dynamic web testing tool, two static source code analyzers, and an origin analysis tool that checked for vulnerabilities in third party libraries and components used to build software. The tools selected for use in the JFAC Enterprise Software Licensing Pilot program were selected in part based on their reputation in the industry, proliferation in the marketplace, and results of prior research and studies. On the whole, the tools proved to be capable enterprise class products as they supported and were implemented on a diverse set of operating systems, programming languages, and platform targets.

The software licenses for these tools were contracted and procured in bulk by JFAC. The JFAC Coordination Center (JFAC-CC), a subcomponent of JFAC, was then responsible for disseminating the individual licenses to each of the product users and also serving as an intermediary between the product vendors and the product users. Licensees were granted a license by request and approval of a representative from their organization. Licensees were a combination of individuals who specifically requested use of the product and those nominated by their organization to be pilot participants. In total, 248 licenses were distributed across the four product offerings. Approximately 66 different programs, projects, or organizations within the DoD received licenses through the JFAC pilot program.

After the licenses had been available to the licensees for nearly a full calendar year, researchers from the Software Engineering Institute (SEI) at Carnegie Mellon University conducted an outreach and survey effort to solicit feedback on user experiences.

Study Design

A formal online survey was developed by researchers at the SEI and distributed to each of the licensees. Additionally, an outreach program was conducted whereby all licensees were personally contacted via email and/or telephone and interviewed for any additional feedback, as well as reminded to complete the online survey. As part of the agreement to use the software license, pilot participants had to agree to complete a survey at the end of the license year.

The survey questions were the same for all four products. The survey questions focused on the technical environment the tool was deployed in, the effectiveness and perceived value of the tool, and the usability of the tool. Usability was measured using a modified version of the System Usability Scale (SUS), a widely used instrument that provides a quick and reliable indicator of usability [Brooke 1996].

49 out of 115 distributed surveys were completed. These 115 survey candidates represented 248 licenses that were distributed during the pilot program. In some cases, the same program, project, or organization was issued licenses for more than one of the Enterprise Software Licensing Pilot program licenses. In these cases, the participant had to fill out one survey for each product that they were licensed. For various reasons, the sample size may be less than 49 in some cases of individual questions examined below. These reasons include applicability of response, lack of response (some questions were optional), or question was not presented to respondent due to conditional survey logic. On the other hand, some survey questions allowed more than one response so the total responses for that question can be higher than the number of respondents.

Impact of Software Assurance Tools on Software Quality

Survey respondents were asked to indicate the total number of lines of code they scanned with each Software Assurance tool. The cumulative number of all lines of code scanned as indicated on the responses was 22,442,902. These lines of code represented 1,391 unique projects or applications. Note that not all respondents answered these two questions. These totals are based on just the respondents who answered these questions.

The actual number of lines of code scanned during the JFAC Enterprise Software Licensing Pilot program can be estimated based on these responses. A simple approximation of the

number of lines of code scanned during the pilot, based on extrapolating known figures, would be 49,181,213. Likewise, the number of projects or applications scanned during the Pilot was approximately 9,121.

Table 1: Projects and Lines of Code Scanned in FY2016 Pilot

	Licenses Issued	Survey Respondents	Reported Projects Scanned (per Respondents)	Reported Lines of Code Scanned (per Respondents)	Approximate Actual Projects Scanned (extrapolated)	Approximate Actual Lines of Code Scanned (extrapolated)
Dynamic Code Analysis Tool	24	11	16	8,029,000	35	17,517,818
Static Code Analysis Tool "A"	31	15	243	9,805,861	502	20,265,446
Static Code Analysis Tool "B"	19	10	521	4,357,041	990	8,278,378
Origin Analysis Tool	174	14	611	251,000	7,594	3,119,571
PILOT TOTALS	**248**	**49**	**1,391**	**22,442,902**	**9,121**	**49,181,213**

Note that some caution should be taken when examining the results for the origin analysis tool in particular for two reasons. First, this product had a much lower percentage of survey respondents per licenses issued, so there is a greater chance for variance between actual figured reports and extrapolated figures. Further, for an original analysis tool, the number of projects scanned is a more relevant scanned than lines of code scanned.

Issue Detection

Survey respondents were asked to identify how many total issues, warnings, and/or vulnerabilities each tool identified. In total, the tools identified 419,189 issues as reported in survey submissions. Extrapolating this figure out across all license holders, the tools likely helped identify 866,697 issues during the Pilot. It is important to remember that not all issues identified by the tools are actual items to be addressed. Potential issues can often be determined to not be applicable for various reasons and the triaging and handling of such issues is another important feature of Software Assurance tools.

Table 2: Issues Detected in FY2016 Pilot

	Issues Detected	Approximate Actual Issues Detected (extrapolated)
Dynamic Code Analysis Tool	102	223
Static Code Analysis Tool "A"	49,838	102,999
Static Code Analysis Tool "B"	363,344	690,354
Origin Analysis Tool	5,905	73,391
PILOT TOTALS	**419,189**	**866,697**

Issue Correction

Survey respondents were asked questions about whether issues, warnings, and/or vulnerabilities discovered using each tool caused them to take corrective actions or make plans for corrective actions. The first question in this area was whether respondents thought that the issues detected were valid issues that need addressed. 100% of respondents using the static code analysis tool "B" and the dynamic code analysis tool thought the discovered issues were valid and needed addressed. Nearly all respondents for the static code analysis tool "A" felt similarly, while just over ½ the respondents for the origin analysis tool thought the discovered issues warranted attention.

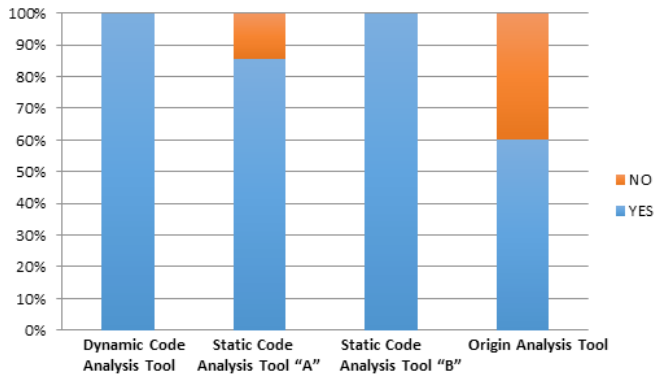


Figure 1: Did the tool find meaningful issues that need addressed?

Respondents were then asked further if they thought the discovered issues were in need of immediate attention, meaning the issues posed a risk of some urgent importance. Nearly ¾ of the dynamic code analysis tool and the static code analysis tool "A" respondents felt some of the detected issues required immediate attention while more than ½ of the static code analysis tools "B" respondents felt the same. 40% of the origin analysis tool respondents thought the detected issues required immediate attention.

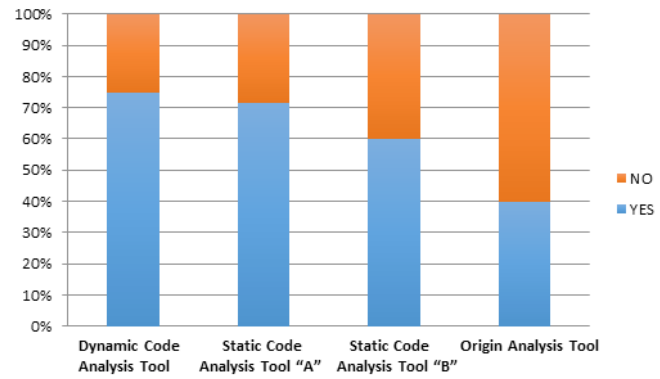


Figure 2: Did the tool find issues that you felt required IMMEDIATE attention?

Having found the detected issues, respondents were asked if they had actually implemented any corrective actions to address these issues. Across all tool products, about 40% of respondents indicated that they had already initiated some corrective action.

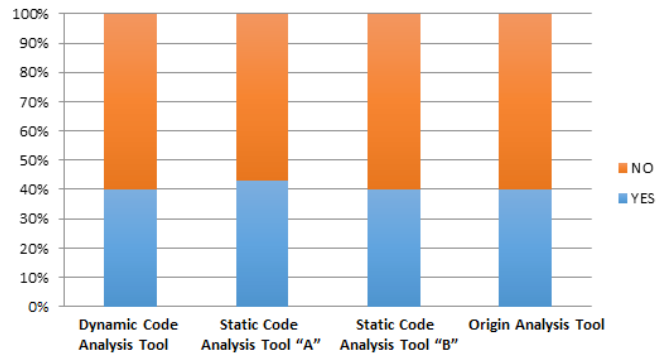


Figure 3: To date, have you fixed/addressed any issues, warnings, and/or vulnerabilities as a result of the tool feedback?

While perhaps not yet initiated, respondents were asked if they had any future plans to implement any corrective actions to address the detected issues. 80% of respondents for the origin analysis tool indicated they had already made plans to correct issues detected and the other 20% indicated they were considering making such plans ("Maybe" response). For the static code analysis tool "B", only 40% of respondents indicated they had already made plans to correct issues detected, while the other 60% indicated they were considering making such plans. For the static code analysis tool "A", just over ½ the respondents indicated they had already made plans to address discovered issues and approximately 30% more indicated they were considering making such plans, while about 15% indicated they had no plans to correct detected issues. Almost exactly ½ of the dynamic code analysis tool respondents indicated they had already made plans to address discovered issues and another ¼ indicated they were considering making such plans, while about 25% indicated they had no plans to correct detected issues.

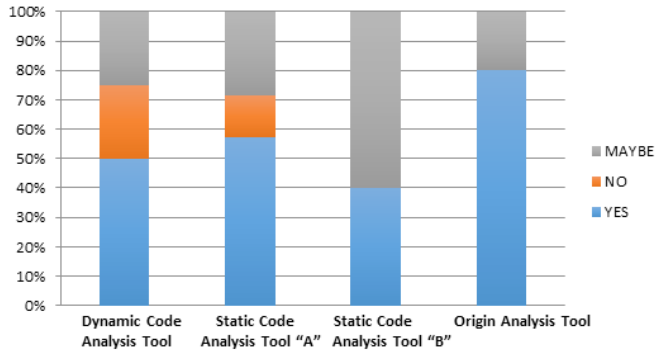


Figure 4: Are there future plans to fix or address any issues, warnings, and/or vulnerabilities as a result of the tool feedback?

Impact of Software Assurance Tools on Development Processes

Beyond making changes to correct specific issues that were detected by each tool, survey respondents were asked whether use of the tools had prompted them to make any changes to their development processes. Across each of the tools, about 20% to 40% of respondents indicated that the tool use had prompted them to make changes in their development process that had already been implemented. About the same number of respondents (20% to 40%) for each tool indicated they had future plans to change their development processes as a result of tool use (Figure 6). However, another 20% to 50% of respondents indicated they were still considering ("Maybe" response) making changes to their development processes as a result of tool use.

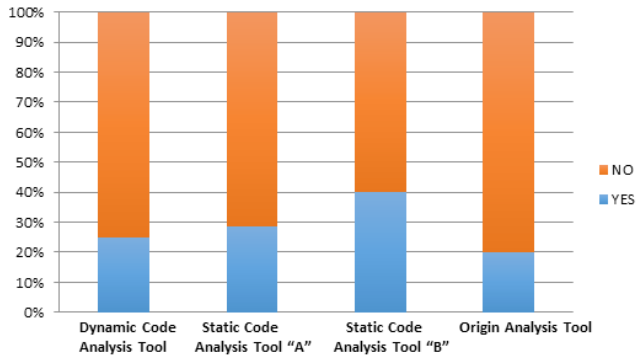


Figure 5: Have you made changes to your design, development, or build processes as result of having used this tool?

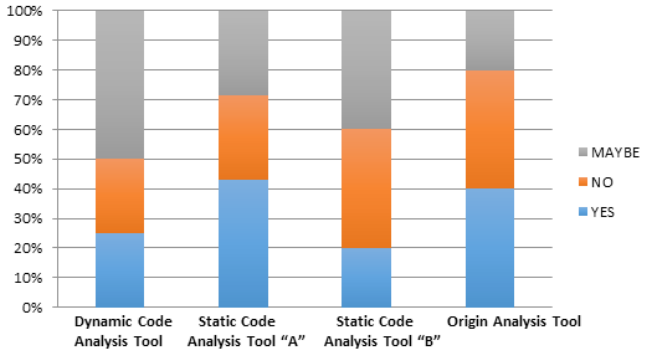


Figure 6: Are there future plans to make changes to your design, development, or build processes as result of having used this tool?

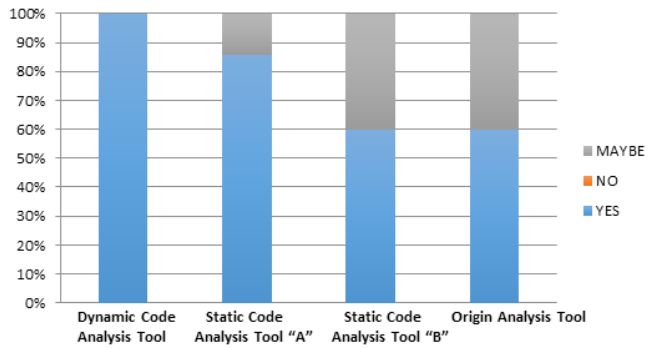


Figure 8: Would you like to continue using this tool in your projects and applications?

Respondents were asked, subjectively, if they felt that each tool is effective at finding meaningful issues, warnings, and/or vulnerabilities in their projects and applications. For the origin analysis tool, the dynamic code analysis tool, and the static code analysis tool "A", roughly 75% of respondents felt that the tool was effective in finding meaningful issues. For the static code analysis tool "B", only 40% of respondents felt the tool was effective, despite it having been reported to find a significant amount of potential vulnerabilities during the Pilot.

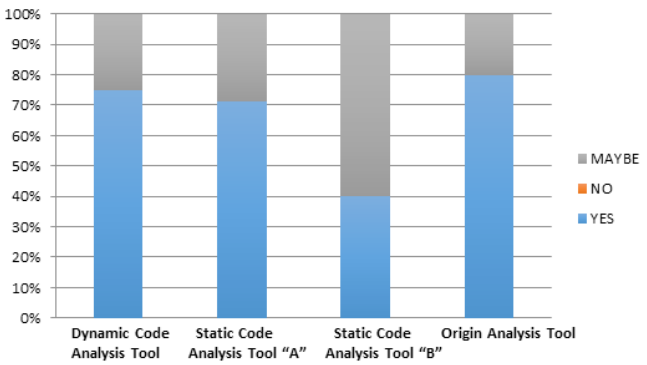


Figure 7: Do you think this tool is effective in finding meaningful issues, warnings, and/or vulnerabilities in your projects and applications?

Respondents were asked whether they would like to continue using each tool in their development and testing processes. 100% of the dynamic code analysis tool respondents indicated that they would like to continue using the tool. For the static code analysis tool "A", 85% of respondents indicated that they definitely would like to continue using the tool, while the other 15% said they "maybe" would like to continue using the tool. For both the static code analysis tool "B" and the origin analysis tool, 60% of respondents indicated that they would like to continue using the tool, while the other 40% of respondents said they "maybe" would like to continue using the tool.

Usability of Software Assurance Tools

Respondents were asked to complete a ten-question questionnaire regarding the usability of each tool. The usability was measured using a modified version of the System Usability Scale [Brooke 1996], which is a commonly used scale for garnering "quick measures" of usability, and is noted to be reliable with small sample sizes.

In general, the average score for a system measured on the SUS is 68. However, it is recommended to normalize scores within a given study to control for environment. In the SUS usability measurements for the Enterprise Software Licensing Pilot program, the dynamic code analysis tool fared the best with an average score of 70.63. The origin analysis tool (61.5) and the static code analysis tool "A" (61.43) had similar average scores, while the static code analysis tool "B" had the lowest average score at 46. Given the prior comment on normalization, the rank order of each product's usability with respect to the other products is more telling than the raw score.

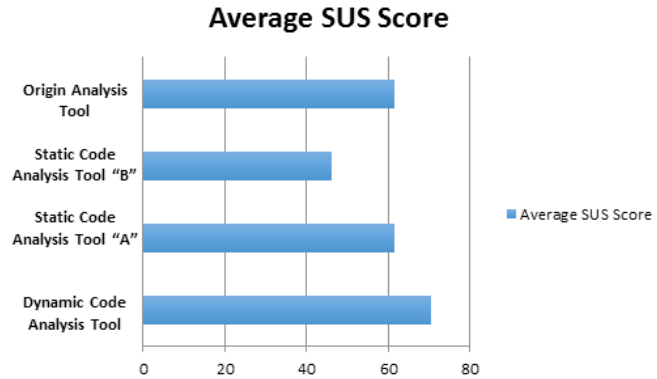


Figure 9: Average SUS usability score for each tool

JFAC Enterprise Software Licensing Program Effectiveness

The JFAC Enterprise Software Licensing Pilot program provided limited quantities of Software Assurance tools to users in the Department of Defense with an aim of evaluating how the use of these tools could improve the state of Software Assurance within the Department. The immediate impact of this pilot was that nearly 50 million lines of code were scanned and approximately 867,000 issues or potential issues were detected. 86% of respondents to the survey felt the issues were meaningful and needed addressed, and more than ½ of respondents thought these detected issues required immediate correction.

So, in a vacuum, the Pilot program achieved success by detecting numerous issues and driving corrective action on a selected set of systems. However, the larger impact delivered by the Pilot is in the way in which the introduction of Software Assurance tools promoted a more mature approach to Software Assurance throughout the development process. More than 25% of survey respondents indicated that they have already made changes to their development and testing process as a result of using the tools in the Pilot program. Another 35% of respondents indicated they are considering making such changes in their development and testing processes. Collectively, more than 50% of the users of the piloted Software Assurance tools not only are making changes to their codebases, but are also making changes to their processes as a result of this program.

Changes and improvements to these processes are likely to yield a much more exponential impact than just the changes to specific lines of code, resulting in greater savings of time, money, and other resources. However, changes in these development processes will require continuous use of Software Assurance tools in the processes. This is reflected in the sentiments of survey respondents as more than 75% of respondents indicated that they felt the tools were useful in finding meaningful issues and more than 75% of respondents wished to continue use of these tools.

While not specifically studied, it is noted anecdotally that no performance issues were reported with the products either. So, in general, the tools proved capable and worthwhile.

Given the effectiveness of this Pilot program, an expansion of the program would have a significant impact on the security of software systems within the DoD. Publicly available data on non-DoD software systems shows that 84% of software breaches exploit vulnerabilities at the application layer [Clark 2015], while funding for information technology (IT) defense versus software assurance is 23-to-1 [Feimann 2014]. It is likely similar numbers exist for DoD software systems and that greater attention to Software Assurance would yield a more mature software security posture for these systems. The potential savings and increase in the programs security posture is significant given the fact that between 1% and 5% of defects discovered in deployment are potentially exploitable vulnerabilities [Woody 2014] and the cost of fixing defects is 10 times more costly to fix after coding and 100 times more costly to fix post deployment [Subramaniam 1999].

Areas for Future Work

There are several areas related to this Pilot program that would be worthwhile areas for further investigation. First, as a substantial number of survey respondents indicated they planned to make changes to their development process as a result of using these tools, it would be beneficial to explore which changes to the process they made or were planning to make. Taking that topic a step further would be to investigate which changes to a development process are most effective for Software Assurance.

A further topic for investigation would be studying the best places in the development process to utilize Software Assurance tools. That is, where should tools be used to get the "best bang for the buck"? Additionally, it should be explored which tool or combination of tools is most effective for certain types of projects and systems.

Lastly, an interesting topic to broach is the impact the usability of a tool has on its perceived value in finding issues. As noted in this study, the static code analysis tool "B" scored very well in terms of finding meaningful/valid issues, yet users were lukewarm on continued use of this product. One reason for this may be the low usability scores the tool received. Is there a correlation between usability and perceived effectiveness and value in Software Assurance tools? Or was this merely a spurious finding? There is a rich canon of study usability of tools and adoption rates, but it would be interesting to explore this relationship specifically for Software Assurance tools.

REFERENCES

[1] Brooke 1996

[2] Brooke, J. (1996). "SUS: a "quick and dirty" usability scale". In P. W. Jordan, B. Thomas, B. A. Weerdmeester, & A. L. McClelland. Usability Evaluation in Industry. London: Taylor and Francis.

[3] Clark 2015

[4] Clark, Tim, Most cyber Attacks Occur from this Common Vulnerability, Forbes. 03-10-2015.

[5] Feiman 2014

[6] Feiman, Joseph, Maverick Research: Stop Protecting Your Apps; It's Time for Apps to Protect Themselves, Gartner. 09-25-2014. G00269825

[7] Subramaniam 1999

[8] Subramaniam, Bala. "Effective Software Defect Tracking Reducing Project Costs and Enhancing Quality," CrossTalk, April 1999: 3-9. JFAC 2016

[9] JFAC. 2016. JFAC Objectives retrieved from JFAC website https://jfac.army.mil on December 20, 2016

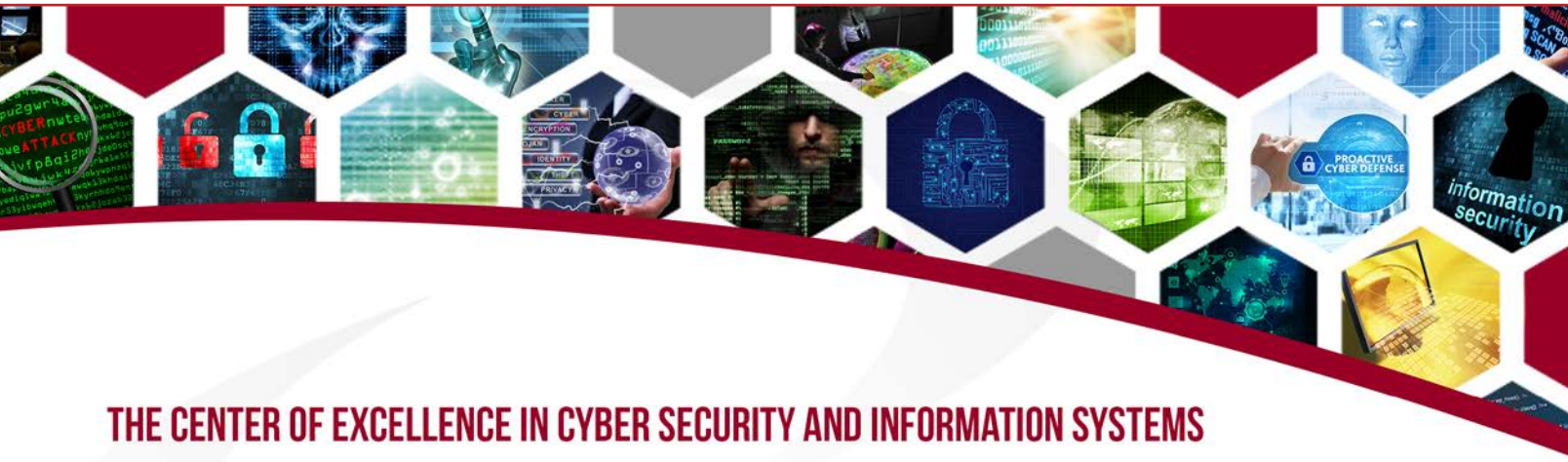
[10] Woody 2014

[11] Woody, Carol, Robert Ellison, and William Nichols. 2014. "Predicting Software Assurance Using Quality and Reliability Measures." CMU/SEI-2014-TN-026. Pittsburgh

Cyber Security and Information Systems
Information Analysis Center
266 Genesee Street
Utica, NY 13502

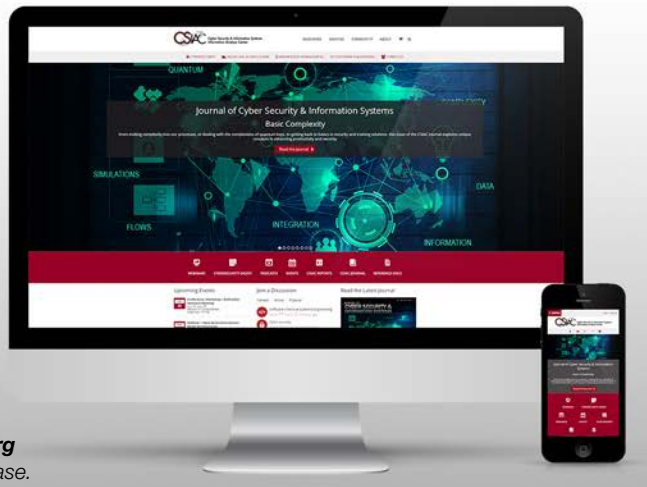
Return Service Requested

PRSRT STD
U.S. Postage
P A I D
Permit #566
UTICA, NY



THE CENTER OF EXCELLENCE IN CYBER SECURITY AND INFORMATION SYSTEMS

Leveraging the best practices and expertise from government, industry, and academia in order to solve your scientific and technical problems



HTTPS://WWW.CSIAC.ORG/JOURNAL/